Java学科面试宝典V7.1

# 一．Java基础

## 1. Java 基础知识

### 1.1 重载和重写的区别

**重载**： 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以

不同，发生在编译时。

**重写**： 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，

访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 则子类就不能重写该方法。

### 1.2 String 和 StringBuffffer、StringBuilder 的区别是什么？String 为什么是不可变的？

**可变性**

简单的来说：String 类中使用 fifinal 关键字字符数组保存字符串， private final char value[] ，所以 String对象是不可变的。而StringBuilder 与 StringBuffffer 都继承自 AbstractStringBuilder 类，在 AbstractStringBuilder中也是使用字符数组保存字符串 char[]value 但是没有用 fifinal 关键字修饰，所以这两种对象都是可变的。

StringBuilder 与 StringBuffffer 的构造方法都是调用父类构造方法也就是 AbstractStringBuilder 实现的，大家可以自

行查阅源码。

AbstractStringBuilder.java

abstract class AbstractStringBuilder implements Appendable, CharSequence {

char[] value;

int count;

AbstractStringBuilder() {

}

AbstractStringBuilder(int capacity) {

value = new char[capacity];

}

**线程安全性**

String 中的对象是不可变的，也就可以理解为常量，线程安全。AbstractStringBuilder 是 StringBuilder 与 StringBuffffer 的公共父类，定义了一些字符串的基本操作，如 expandCapacity、append、insert、indexOf 等公共 方法。StringBuffffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对

方法进行加同步锁，所以是非线程安全的。

**性能**

abstract class AbstractStringBuilder implements Appendable, CharSequence {

char[] value;

int count;

AbstractStringBuilder() {

}

AbstractStringBuilder(int capacity) {

value = new char[capacity];

}每次对 String 类型进行改变的时候，都会生成一个新的 String 对象，然后将指针指向新的 String 对象。

StringBuffffer 每次都会对 StringBuffffer 对象本身进行操作，而不是生成新的对象并改变对象引用。相同情况下使用 StirngBuilder 相比使用 StringBuffffer 仅能获得 10%~15% 左右的性能提升，但却要冒多线程不安全的风险。

**对于三者使用的总结：**

1. 操作少量的数据 = String

2. 单线程操作字符串缓冲区下操作大量数据 = StringBuilder

3. 多线程操作字符串缓冲区下操作大量数据 = StringBuffffer

### 1.3 自动装箱与拆箱

**装箱**：将基本类型用它们对应的引用类型包装起来；

**拆箱**：将包装类型转换为基本数据类型；

### 1.4 == 与 equals

**==** : 它的作用是判断两个对象的地址是不是相等。即，判断两个对象是不是同一个对象。(基本数据类型==比较的是值，引用数据类型==比较的是内存地址)

**equals()** : 它的作用也是判断两个对象是否相等。但它一般有两种使用情况：

情况1：类没有覆盖 equals() 方法。则通过 equals() 比较该类的两个对象时，等价于通过“==”比较这两个对象。

情况2：类覆盖了 equals() 方法。一般，我们都覆盖 equals() 方法来两个对象的内容相等；若它们的内容相等，则返回 true (即，认为这两个对象相等)。

public class test1 {

public static void main(String[] args) {

String a = new String("ab"); // a 为一个引用

String b = new String("ab"); // b为另一个引用,对象的内容一样

String aa = "ab"; // 放在常量池中

String bb = "ab"; // 从常量池中查找

if (aa == bb) // true

System.out.println("aa==bb");

if (a == b) // false，非同一对象

System.out.println("a==b");

if (a.equals(b)) // true

System.out.println("aEQb");

if (42 == 42.0) { // true

System.out.println("true");

}

}

}

**说明：**

String 中的 equals 方法是被重写过的，因为 object 的 equals 方法是比较的对象的内存地址，而 String 的equals 方法比较的是对象的值。

当创建 String 类型的对象时，虚拟机会在常量池中查找有没有已经存在的值和要创建的值相同的对象，如果有就把它赋给当前引用。如果没有就在常量池中重新创建一个 String 对象。

### 1.5 关于 fifinal 关键字的一些总结

fifinal关键字主要用在三个地方：变量、方法、类。

1. 对于一个fifinal变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

2. 当用fifinal修饰一个类时，表明这个类不能被继承。fifinal类中的所有成员方法都会被隐式地指定为fifinal方法。

3. 使用fifinal方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。

在早期的Java实现版本中，会将fifinal方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升（现在的Java版本已经不需要使用fifinal方法进行这些优化了）。类中所有的private方法都隐式地指定为fifianl。

### 1.6 Object类的常见方法总结

Object类是一个特殊的类，是所有类的父类。它主要提供了以下11个方法：

public final native Class<?> getClass()//native方法，用于返回当前运行时对象的Class对象，使用了

final关键字修饰，故不允许子类重写。

public native int hashCode() //native方法，用于返回对象的哈希码，主要使用在哈希表中，比如JDK中的

HashMap。

public boolean equals(Object obj)//用于比较2个对象的内存地址是否相等，String类对该方法进行了重写用户

比较字符串的值是否相等。

protected native Object clone() throws CloneNotSupportedException//naitive方法，用于创建并返回

当前对象的一份拷贝。一般情况下，对于任何对象 x，表达式 x.clone() != x 为true，x.clone().getClass()

== x.getClass() 为true。Object本身没有实现Cloneable接口，所以不重写clone方法并且进行调用的话会发生

CloneNotSupportedException异常。

public String toString()//返回类的名字@实例的哈希码的16进制的字符串。建议Object所有的子类都重写这个方

法。

public final native void notify()//native方法，并且不能重写。唤醒一个在此对象监视器上等待的线程(监视

器相当于就是锁的概念)。如果有多个线程在等待只会任意唤醒一个。

public final native void notifyAll()//native方法，并且不能重写。跟notify一样，唯一的区别就是会唤醒

在此对象监视器上等待的所有线程，而不是一个线程。

public final native void wait(long timeout) throws InterruptedException//native方法，并且不能

重写。暂停线程的执行。注意：sleep方法没有释放锁，而wait方法释放了锁 。timeout是等待时间。

public final void wait(long timeout, int nanos) throws InterruptedException//多了nanos参数，

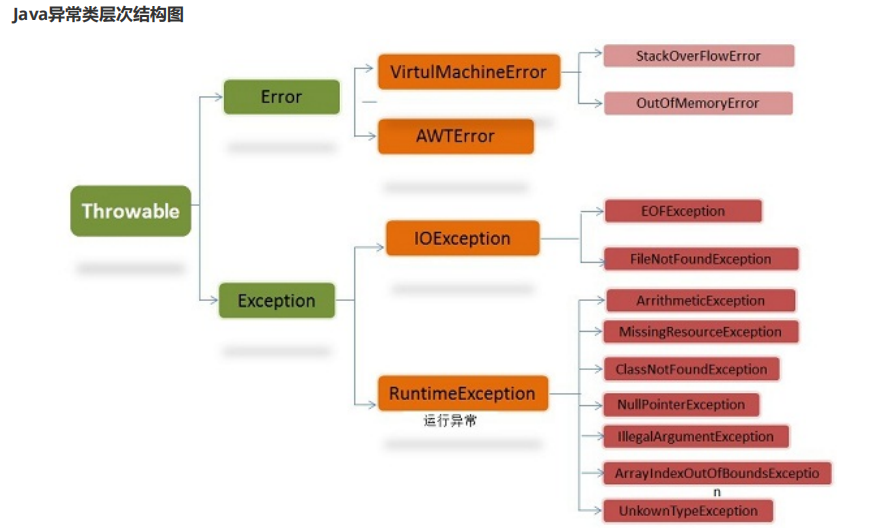
这个参数表示额外时间（以毫微秒为单位，范围是 0-999999）。 所以超时的时间还需要加上nanos毫秒。

public final void wait() throws InterruptedException//跟之前的2个wait方法一样，只不过该方法一直等

待，没有超时时间这个概念

protected void finalize() throws Throwable { }//实例被垃圾回收器回收的时候触发的操作

### 1.7 Java 中的异常处理



在 Java 中，所有的异常都有一个共同的祖先java.lang包中的 **Throwable类**。Throwable： 有两个重要的子类：

**Exception（异常）** 和 **Error（错误）** ，二者都是 Java 异常处理的重要子类，各自都包含大量子类。

**Error（错误）:是程序无法处理的错误**，表示运行应用程序中较严重问题。大多数错误与代码编写者执行的操作无关，而表示代码运行时 JVM（Java 虚拟机）出现的问题。例如，Java虚拟机运行错误（Virtual MachineError），当JVM 不再有继续执行操作所需的内存资源时，将出现 OutOfMemoryError。这些异常发生时，Java虚拟机（JVM）一般会选择线程终止。

这些错误表示故障发生于虚拟机自身、或者发生在虚拟机试图执行应用时，如Java虚拟机运行错误（Virtual MachineError）、类定义错误（NoClassDefFoundError）等。这些错误是不可查的，因为它们在应用程序的控制和处理能力之 外，而且绝大多数是程序运行时不允许出现的状况。对于设计合理的应用程序来说，即使确实发生了错

误，本质上也不应该试图去处理它所引起的异常状况。在 Java中，错误通过Error的子类描述。

**Exception（异常）:是程序本身可以处理的异常**。Exception 类有一个重要的子类 **RuntimeException**。

RuntimeException 异常由Java虚拟机抛出。**NullPointerException**（要访问的变量没有引用任何对象时，抛出该 异常）、**ArithmeticException**（算术运算异常，一个整数除以0时，抛出该异常）和

**ArrayIndexOutOfBoundsException** （下标越界异常）。

**注意：异常和错误的区别：异常能被程序本身可以处理，错误是无法处理。**

**Throwable类常用方法**

**public string getMessage()**:返回异常发生时的详细信息

**public string toString()**:返回异常发生时的简要描述

**public string getLocalizedMessage()**:返回异常对象的本地化信息。使用Throwable的子类覆盖这个方法，可以声称本地化信息。如果子类没有覆盖该方法，则该方法返回的信息与getMessage（）返回的结果相同

**public void printStackTrace()**:在控制台上打印Throwable对象封装的异常信息

**异常处理总结**

try 块：用于捕获异常。其后可接零个或多个catch块，如果没有catch块，则必须跟一个fifinally块。catch 块：用于处理try捕获到的异常。

fifinally 块：无论是否捕获或处理异常，fifinally块里的语句都会被执行。当在try块或catch块中遇到return语句

时，fifinally语句块将在方法返回之前被执行。

**在以下4种特殊情况下，fifinally块不会被执行：**

1. 在fifinally语句块中发生了异常。

2. 在前面的代码中用了System.exit()退出程序。

3. 程序所在的线程死亡。

4. 关闭CPU。

### 1.8 获取用键盘输入常用的的两种方法

**方法1：通过 Scanner**

Scanner input = new Scanner(System.in);

String s = input.nextLine();

input.close();

**方法2：通过 BufffferedReader**

BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

String s = input.readLine();

### 1.9 接口和抽象类的区别是什么

1. 接口的方法默认是 public，所有方法在接口中不能有实现(Java 8 开始接口方法可以有默认实现），抽象类可以有非抽象的方法

2. 接口中的实例变量默认是 final 类型的，而抽象类中则不一定

3. 一个类可以实现多个接口，但最多只能实现一个抽象类

4. 一个类实现接口的话要实现接口的所有方法，而抽象类不一定

5. 接口不能用 new 实例化，但可以声明，但是必须引用一个实现该接口的对象 从设计层面来说，抽象是对类的抽象，是一种模板设计，接口是行为的抽象，是一种行为的规范。

备注:在JDK8中，接口也可以定义静态方法，可以直接用接口名调用。实现类和实现是不可以调用的。如果同时实现两个接口，接口中定义了一样的默认方法，必须重写，不然会报错。

## 2.Java 集合框架

### 2.1 Arraylist 与 LinkedList 异同

**1. 是否保证线程安全：** ArrayList 和 LinkedList 都是不同步的，也就是不保证线程安全；

**2. 底层数据结构：** Arraylist 底层使用的是Object数组；LinkedList 底层使用的是双向链表数据结构（JDK1.6之 前为循环链表，JDK1.7取消了循环。注意双向链表和双向循环链表的区别：）；

**3. 插入和删除是否受元素位置的影响：** ① **ArrayList 采用数组存储，所以插入和删除元素的时间复杂度受元素**

**位置的影响。** 比如：执行 add(E e) 方法的时候， ArrayList 会默认在将指定的元素追加到此列表的末尾，这种情况时间复杂度就是O(1)。但是如果要在指定位置 i 插入和删除元素的话（ add(int index, E element) ）时间复杂度就为 O(n-i)。因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的(n-i)个元素都要执行向后位/向前移一位的操作。 ② **LinkedList 采用链表存储，所以插入，删除元素时间复杂度不受元素位置的影响，都是近似 O（1）而数组为近似 O（n）。**

**4. 是否支持快速随机访问：** LinkedList 不支持高效的随机元素访问，而 ArrayList 支持。快速随机访问就是通过元素的序号快速获取元素对象(对应于 get(int index) 方法)。

**5. 内存空间占用：** ArrayList的空 间浪费主要体现在在list列表的结尾会预留一定的容量空间，而LinkedList的空间花费则体现在它的每一个元素都需要消耗比ArrayList更多的空间（因为要存放直接后继和直接前驱以及数据）。

**补充内容:RandomAccess接口**

public interface RandomAccess {

}

查看源码我们发现实际上 RandomAccess 接口中什么都没有定义。所以，在我看来 RandomAccess 接口不过是一个标识罢了。标识什么？ 标识实现这个接口的类具有随机访问功能。

在binarySearch（）方法中，它要判断传入的list 是否RamdomAccess的实例，如果是，调用

indexedBinarySearch（）方法，如果不是，那么调用iteratorBinarySearch（）方法

public static <T>

int binarySearch(List<? extends Comparable<? super T>> list, T key) {

if (list instanceof RandomAccess || list.size()<BINARYSEARCH\_THRESHOLD)

return Collections.indexedBinarySearch(list, key);

else

return Collections.iteratorBinarySearch(list, key);

}

ArrayList 实现了 RandomAccess 接口， 而 LinkedList 没有实现。为什么呢？我觉得还是和底层数据结构有关！ArrayList 底层是数组，而 LinkedList 底层是链表。数组天然支持随机访问，时间复杂度为 O（1），所以称为快速随机访问。链表需要遍历到特定位置才能访问特定位置的元素，时间复杂度为 O（n），所以不支持快速随机访问。，

ArrayList 实现了 RandomAccess 接口，就表明了他具有快速随机访问功能。 RandomAccess 接口只是标识，并不是说 ArrayList 实现 RandomAccess 接口才具有快速随机访问功能的！

**下面再总结一下 list 的遍历方式选择：**

实现了RandomAccess接口的list，优先选择普通for循环 ，其次foreach,未实现RandomAccess接口的list， 优先选择iterator遍历（foreach遍历底层也是通过iterator实现的），大size的数据，千万不要使用普通for循环.补充：数据结构基础之双向链表

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。一般我们都构造双向循环链表，如

下图所示，同时下图也是LinkedList 底层使用的是双向循环链表数据结构。

public interface RandomAccess {

}

public static <T>

int binarySearch(List<? extends Comparable<? super T>> list, T key) {

if (list instanceof RandomAccess || list.size()<BINARYSEARCH\_THRESHOLD)

return Collections.indexedBinarySearch(list, key);

else

return Collections.iteratorBinarySearch(list, key);

}



### 2.2 ArrayList 与 Vector 区别

Vector类的所有方法都是同步的。可以由两个线程安全地访问一个Vector对象、但是一个线程访问Vector的话代码要在同步操作上耗费大量的时间。

Arraylist不是同步的，所以在不需要保证线程安全时时建议使用Arraylist。

### 2.3 HashMap的底层实现

**JDK1.8之前**

JDK1.8 之前 HashMap 底层是 **数组和链表** 结合在一起使用也就是 **链表散列**。**HashMap 通过 key 的 hashCode 经过扰动函数处理过后得到 hash 值，然后通过** (n **-** 1) & hash **判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。**

**所谓扰动函数指的就是 HashMap 的 hash 方法。使用 hash 方法也就是扰动函数是为了防止一些实现比较差的hashCode() 方法 换句话说使用扰动函数之后可以减少碰撞。**

**JDK 1.8 HashMap 的 hash 方法源码:**

JDK 1.8 的 hash方法 相比于 JDK 1.7 hash 方法更加简化，但是原理不变。

static final int hash(Object key) {

int h;

// key.hashCode()：返回散列值也就是hashcode

// ^ ：按位异或

// >>>:无符号右移，忽略符号位，空位都以0补齐

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

对比一下 JDK1.7的 HashMap 的 hash 方法源码.

static int hash(int h) {

// This function ensures that hashCodes that differ only by

// constant multiples at each bit position have a bounded

// number of collisions (approximately 8 at default load factor).

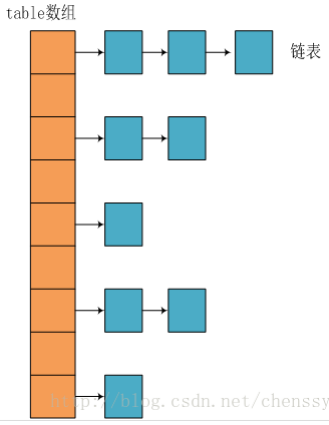
h ^= (h >>> 20) ^ (h >>> 12);

return h ^ (h >>> 7) ^ (h >>> 4);

}

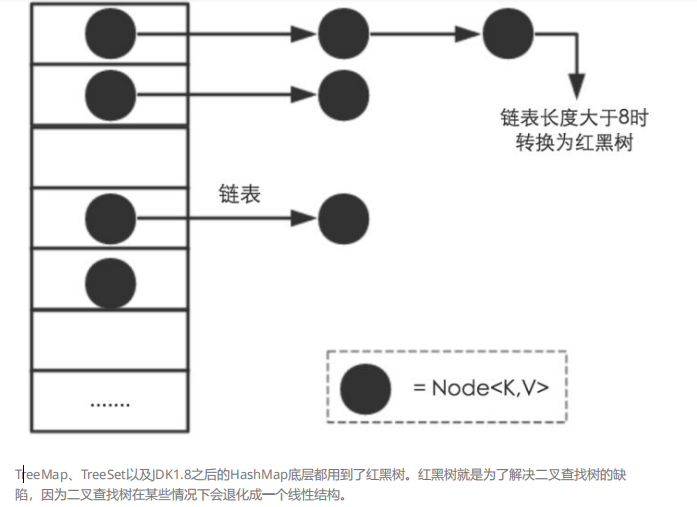
相比于 JDK1.8 的 hash 方法 ，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次。所谓 **“拉链法”** 就是：将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希

冲突，则将冲突的值加到链表中即可。



**JDK1.8之后**

相比于之前的版本， JDK1.8之后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。



### 2.4 HashMap 和 Hashtable 的区别

1. **线程是否安全：** HashMap 是非线程安全的，HashTable 是线程安全的；HashTable 内部的方法基本都经过

synchronized 修饰。（如果你要保证线程安全的话就使用 ConcurrentHashMap 吧！）；

2. **效率：** 因为线程安全的问题，HashMap 要比 HashTable 效率高一点。另外，HashTable 基本被淘汰，不要在代码中使用它；

3. **对Null key 和Null value的支持：** HashMap 中，null 可以作为键，这样的键只有一个，可以有一个或多个键

所对应的值为 null。。但是在 HashTable 中 put 进的键值只要有一个 null，直接抛出 NullPointerException。

4. **初始容量大小和每次扩充容量大小的不同 ：** ①创建时如果不指定容量初始值，Hashtable 默认的初始大小为11，之后每次扩充，容量变为原来的2n+1。HashMap 默认的初始化大小为16。之后每次扩充，容量变为原来的2倍。②创建时如果给定了容量初始值，那么 Hashtable 会直接使用你给定的大小，而 HashMap 会将其扩充为2的幂次方大小（HashMap 中的 tableSizeFor() 方法保证，下面给出了源代码）。也就是说 HashMap 总是使用2的幂作为哈希表的大小,后面会介绍到为什么是2的幂次方。

5. **底层数据结构：** JDK1.8 以后的 HashMap 在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。Hashtable 没有这样的机制。

**HasMap 中带有初始容量的构造函数：**

public HashMap(int initialCapacity, float loadFactor) {

if (initialCapacity < 0)

throw new IllegalArgumentException("Illegal initial capacity: " +

initialCapacity);

if (initialCapacity > MAXIMUM\_CAPACITY)

initialCapacity = MAXIMUM\_CAPACITY;

if (loadFactor <= 0 || Float.isNaN(loadFactor))

throw new IllegalArgumentException("Illegal load factor: " +

loadFactor);

this.loadFactor = loadFactor;

this.threshold = tableSizeFor(initialCapacity);

}

public HashMap(int initialCapacity) {

this(initialCapacity, DEFAULT\_LOAD\_FACTOR);

}

下面这个方法保证了 HashMap 总是使用2的幂作为哈希表的大小。

/\*\*

\* Returns a power of two size for the given target capacity.

\*/

static final int tableSizeFor(int cap) {

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

**2.5 HashMap 的长度为什么是2的幂次方**

为了能让 HashMap 存取高效，尽量较少碰撞，也就是要尽量把数据分配均匀。我们上面也讲到了过了，Hash 值的范围值-2147483648到2147483647，前后加起来大概40亿的映射空间，只要哈希函数映射得比较均匀松散，一般应用是很难出现碰撞的。但问题是一个40亿长度的数组，内存是放不下的。所以这个散列值是不能直接拿来用的。用之

前还要先做对数组的长度取模运算，得到的余数才能用来要存放的位置也就是对应的数组下标。这个数组下标的计算方法是“ (n - 1) & hash ”。（n代表数组长度）。这也就解释了 HashMap 的长度为什么是2的幂次方。

**这个算法应该如何设计呢？**

我们首先可能会想到采用%取余的操作来实现。但是，重点来了：**“取余(%)操作中如果除数是2的幂次则等价于与其**

**除数减一的与(&)操作（也就是说 hash%length==hash&(length-1)的前提是 length 是2的 n 次方；）。”** 并且 **采**

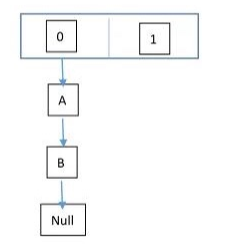
**用二进制位操作 &，相对于%能够提高运算效率，这就解释了 HashMap 的长度为什么是2的幂次方。**

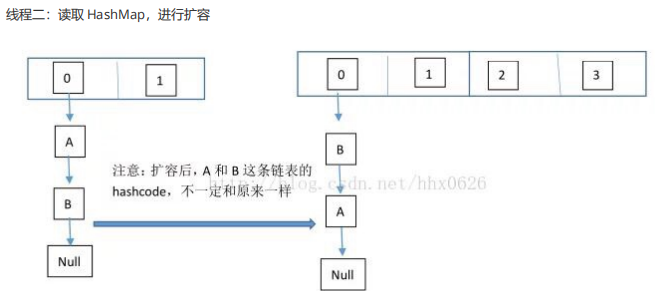
### 2.6 HashMap 多线程操作导致死循环问题

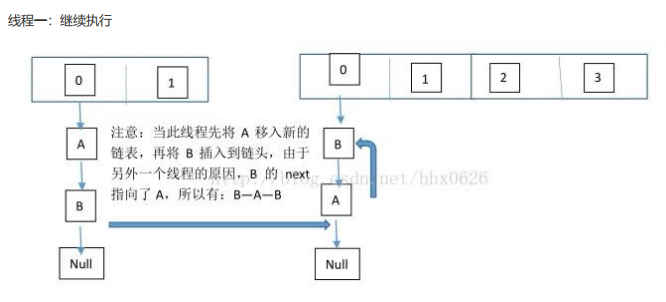
在多线程下，进行 put 操作会导致 HashMap 死循环，原因在于 HashMap 的扩容 resize()方法。由于扩容是新建一个数组，复制原数据到数组。由于数组下标挂有链表，所以需要复制链表，但是多线程操作有可能导致环形链表。复制链表过程如下:

以下模拟2个线程同时扩容。假设，当前 HashMap 的空间为2（临界值为1），hashcode 分别为 0 和 1，在散列地 址 0 处有元素 A 和 B，这时候要添加元素 C，C 经过 hash 运算，得到散列地址为 1，这时候由于超过了临界值，空间不够，需要调用 resize 方法进行扩容，那么在多线程条件下，会出现条件竞争，模拟过程如下：

线程一：读取到当前的 HashMap 情况，在准备扩容时，线程二介入







这个过程为，先将 A 复制到新的 hash 表中，然后接着复制 B 到链头（A 的前边：B.next=A），本来 B.next=null，到此也就结束了（跟线程二一样的过程），但是，由于线程二扩容的原因，将 B.next=A，所以，这里继续复制A，让A.next=B，由此，环形链表出现：B.next=A; A.next=B

**注意：jdk1.8已经解决了死循环的问题。**

### 2.7 HashSet 和 HashMap 区别

如果你看过 HashSet 源码的话就应该知道：HashSet 底层就是基于 HashMap 实现的。（HashSet 的源码非常非常少，因为除了 clone() 方法、writeObject()方法、readObject()方法是 HashSet 自己不得不实现之外，其他方法都是直接调用 HashMap 中的方法。）



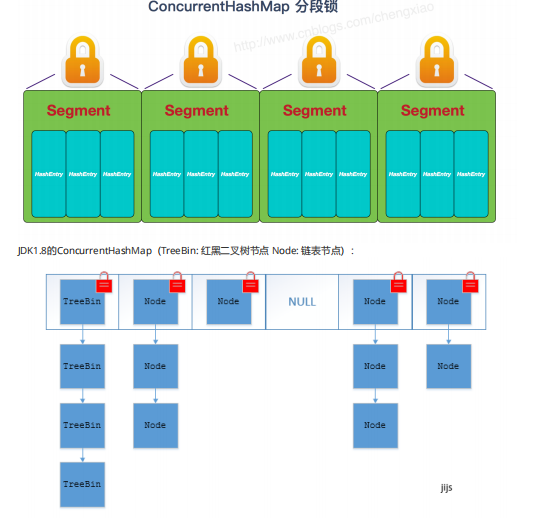
### 2.8 ConcurrentHashMap 和 Hashtable 的区别

ConcurrentHashMap 和 Hashtable 的区别主要体现在实现线程安全的方式上不同。

**底层数据结构：** JDK1.7的 ConcurrentHashMap 底层采用 **分段的数组+链表** 实现，JDK1.8 采用的数据结构跟HashMap1.8的结构一样，数组+链表/红黑二叉树。Hashtable 和 JDK1.8 之前的 HashMap 的底层数据结构类似都是采用 **数组+链表** 的形式，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的；

**实现线程安全的方式（重要）：** ① **在JDK1.7的时候，ConcurrentHashMap（分段锁）** 对整个桶数组进行了分割分段(Segment)，每一把锁只锁容器其中一部分数据，多线程访问容器里不同数据段的数据，就不会存在锁竞争，提高并发访问率。 **到了 JDK1.8 的时候已经摒弃了Segment的概念，而是直接用 Node 数组+链表+红黑树的数据结构来实现，并发控制使用 synchronized 和 CAS 来操作。（JDK1.6以后 对 synchronized锁做了很多优化）** 整个看起来就像是优化过且线程安全的 HashMap，虽然在JDK1.8中还能看到 Segment 的数据结构,但是已经简化了属性，只是为了兼容旧版本；② **Hashtable(同一把锁)** :使用 synchronized 来保证线程安全,效率非常低下。当一个线程访问同步方法时，其他线程也访问同步方法，可能会进入阻塞或轮询状态，如使用put 添加元素，另一个线程不能使用 put 添加元素，也不能使用 get，竞争会越来越激烈效率越低。





### 2.9 ConcurrentHashMap线程安全的具体实现方式/底层具体实现

**JDK1.7（上面有示意图）**

首先将数据分为一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据时，其他段的数据也能被其他线程访问。

**ConcurrentHashMap 是由 Segment 数组结构和 HashEntry 数组结构组成**。

Segment 实现了 ReentrantLock,所以 Segment 是一种可重入锁，扮演锁的角色。HashEntry 用于存储键值对数据。

static class Segment<K,V> extends ReentrantLock implements Serializable {

}

一个 ConcurrentHashMap 里包含一个 Segment 数组。Segment 的结构和HashMap类似，是一种数组和链表结构，一个 Segment 包含一个 HashEntry 数组，每个 HashEntry 是一个链表结构的元素，每个 Segment 守护着一个HashEntry数组里的元素，当对 HashEntry 数组的数据进行修改时，必须首先获得对应的 Segment的锁。

**JDK1.8 （上面有示意图）**

ConcurrentHashMap取消了Segment分段锁，采用CAS和synchronized来保证并发安全。数据结构跟HashMap1.8的结构类似，数组+链表/红黑二叉树。

synchronized只锁定当前链表或红黑二叉树的首节点，这样只要hash不冲突，就不会产生并发，效率又提升N倍。

### 2.10 集合框架底层数据结构总结

**Collection**

**1. List**

**Arraylist：** Object数组

**Vector：** Object数组

**LinkedList：** 双向链表(JDK1.6之前为循环链表，JDK1.7取消了循环) 详细可阅读JDK1.7-LinkedList循环链表优化

**2. Set**

**HashSet（无序，唯一）:** 基于 HashMap 实现的，底层采用 HashMap 来保存元素

**LinkedHashSet：** LinkedHashSet 继承与 HashSet，并且其内部是通过 LinkedHashMap 来实现的。有点类

似于我们之前说的LinkedHashMap 其内部是基于 Hashmap 实现一样，不过还是有一点点区别的。

**TreeSet（有序，唯一）：** 红黑树(自平衡的排序二叉树。)

**Map**

**HashMap：** JDK1.8之前HashMap由数组+链表组成的，数组是HashMap的主体，链表则是主要为了解决哈希冲突而存在的（“拉链法”解决冲突）.JDK1.8以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间

**LinkedHashMap:** LinkedHashMap 继承自 HashMap，所以它的底层仍然是基于拉链式散列结构即由数组和链表或红黑树组成。另外，LinkedHashMap 在上面结构的基础上，增加了一条双向链表，使得上面的结构可以保持键值对的插入顺序。同时通过对链表进行相应的操作，实现了访问顺序相关逻辑。详细可以查看：

**HashTable:** 数组+链表组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的

**TreeMap:** 红黑树（自平衡的排序二叉树）

## 3.Java多线程

关于 Java多线程，在面试的时候，问的比较多的就是**①悲观锁和乐观锁**（ 具体可以看我的这篇文章：面试必备之乐观锁与悲观锁）、**②synchronized和lock区别以及volatile和synchronized的区别**，**③可重入锁与非可重入锁的区别**、**④多线程是解决什么问题的**、**⑤线程池解决什么问题**、**⑥线程池的原理**、**⑦线程池使用时的注意事项**、**⑧AQS原有没有在项目中实际使用多线程的经历**。所以，如果你在你的项目中有实际使用Java多线程的经历 的话，会为你加分不少哦！

### 3.1 面试中关于 synchronized 关键字的 5 连击

**3.1.1 说一说自己对于 synchronized 关键字的了解**

synchronized关键字解决的是多个线程之间访问资源的同步性，synchronized关键字可以保证被它修饰的方法或者代码块在任意时刻只能有一个线程执行。 另外，在 Java 早期版本中，synchronized属于重量级锁，效率低下，因为监视器锁（monitor）是依赖于底层的操作系统的 Mutex Lock 来实现的，Java 的线程是映射到操作系统的原生线程之上的。如果要挂起或者唤醒一个线程，都需要操作系统帮忙完成，而操作系统实现线程之间的切换时需要从用户态转换到内核态，这个状态之间的转换需要相对比较长的时间，时间成本相对较高，这也是为什么早期的 synchronized 效率低的原因。庆幸的是在 Java 6 之后Java 官方对从 JVM 层面对synchronized 较大优化，所以现在的 synchronized 锁效率也优化得很不错了。JDK1.6对锁的实现引入了大量的优化，如自旋锁、适应性自旋锁、锁消除、锁粗化、偏向锁、轻量级锁等技术来减少锁操作的开销。

**3.1.2 说说自己是怎么使用 synchronized 关键字，在项目中用到了吗**

**synchronized关键字最主要的三种使用方式：**

**修饰实例方法，作用于当前对象实例加锁，进入同步代码前要获得当前对象实例的锁**

**修饰静态方法，作用于当前类对象加锁，进入同步代码前要获得当前类对象的锁** 。也就是给当前类加锁，会作用于类的所有对象实例，因为静态成员不属于任何一个实例对象，是类成员（ static 表明这是该类的一个静态资源，不管new了多少个对象，只有一份，所以对该类的所有对象都加了锁）。所以如果一个线程A调用一个实例对象的非静态 synchronized 方法，而线程B需要调用这个实例对象所属类的静态 synchronized 方法，是允许的，不会发生互斥现象，**因为访问静态 synchronized 方法占用的锁是当前类的锁，而访问非静态 synchronized 方法占用的锁是当前实例对象锁**。

**修饰代码块，指定加锁对象，对给定对象加锁，进入同步代码库前要获得给定对象的锁。** 和 synchronized 方法一样，synchronized(this)代码块也是锁定当前对象的。synchronized 关键字加到 static 静态方法和 synchronized(class)代码块上都是是给 Class 类上锁。这里再提一下：synchronized关键字加到非 static 静态方法上是给对象实例上锁。另外需要注意的是：尽量不要使用 synchronized(String a) 因为JVM中，字符串常量

池具有缓冲功能！下面我已一个常见的面试题为例讲解一下 synchronized 关键字的具体使用。

面试中面试官经常会说：“单例模式了解吗？来给我手写一下！给我解释一下双重检验锁方式实现单例模式的原理呗！”

**双重校验锁实现对象单例（线程安全）**

public class Singleton {

private volatile static Singleton uniqueInstance;

private Singleton() {

}

public static Singleton getUniqueInstance() {

//先判断对象是否已经实例过，没有实例化过才进入加锁代码另外，需要注意 uniqueInstance 采用 volatile 关键字修饰也是很有必要。

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分

为三步执行：

1. 为 uniqueInstance 分配内存空间

2. 初始化 uniqueInstance

3. 将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1->3->2。指令重排在单线程环境下不会出先问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被

初始化。使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行。

**3.1.3 讲一下 synchronized 关键字的底层原理**

**synchronized 关键字底层原理属于 JVM 层面。**

**① synchronized 同步语句块的情况**

通过 JDK 自带的 javap 命令查看 SynchronizedDemo 类的相关字节码信息：首先切换到类的对应目录执行 javacSynchronizedDemo.java 命令生成编译后的 .class 文件，然后执行 javap -c -s -v -l SynchronizedDemo.class 。

if (uniqueInstance == null) {

//类对象加锁

synchronized (Singleton.class) {

if (uniqueInstance == null) {

uniqueInstance = new Singleton();

}

}

}

return uniqueInstance;

}

}

public class SynchronizedDemo {

public void method() {

synchronized (this) {

System.out.println("synchronized 代码块");

}

}

}从上面我们可以看出：

**synchronized 同步语句块的实现使用的是 monitorenter 和 monitorexit 指令，其中 monitorenter 指令指向同步代码块的开始位置，monitorexit 指令则指明同步代码块的结束位置。** 当执行 monitorenter 指令时，线程试图获取锁也就是获取 monitor(monitor对象存在于每个Java对象的对象头中，synchronized 锁便是通过这种方式获取锁的，也是为什么Java中任意对象可以作为锁的原因) 的持有权.当计数器为0则可以成功获取，获取后将锁计数器设为1也就是加1。相应的在执行 monitorexit 指令后，将锁计数器设为0，表明锁被释放。如果获取对象锁失败，那当

前线程就要阻塞等待，直到锁被另外一个线程释放为止。

**② synchronized 修饰方法的的情况**

public class SynchronizedDemo2 {

public synchronized void method() {

System.out.println("synchronized 方法");

}

}synchronized 修饰的方法并没有 monitorenter 指令和 monitorexit 指令，取得代之的确实是

ACC\_SYNCHRONIZED 标识，该标识指明了该方法是一个同步方法，JVM 通过该 ACC\_SYNCHRONIZED 访问标志来辨别一个方法是否声明为同步方法，从而执行相应的同步调用。

**3.1.4 说说 JDK1.6 之后的synchronized 关键字底层做了哪些优化，可以详细介绍一下这些优**

**化吗**

JDK1.6 对锁的实现引入了大量的优化，如偏向锁、轻量级锁、自旋锁、适应性自旋锁、锁消除、锁粗化等技术来减少锁操作的开销。

锁主要存在四种状态，依次是：无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态，他们会随着竞争的激烈而逐渐升级。注意锁可以升级不可降级，这种策略是为了提高获得锁和释放锁的效率。

**3.1.5 谈谈 synchronized和ReenTrantLock 的区别**

**① 两者都是可重入锁**

两者都是可重入锁。“可重入锁”概念是：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时这个对象锁还没有释放，当其再次想要获取这个对象的锁的时候还是可以获取的，如果不可锁重入的话，就会造成死锁。同一个线程每次获取锁，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

**② synchronized 依赖于 JVM 而 ReenTrantLock 依赖于 API**

synchronized 是依赖于 JVM 实现的，前面我们也讲到了 虚拟机团队在 JDK1.6 为 synchronized 关键字进行了很多优化，但是这些优化都是在虚拟机层面实现的，并没有直接暴露给我们。ReenTrantLock 是 JDK 层面实现的（也就是 API 层面，需要 lock() 和 unlock 方法配合 try/fifinally 语句块来完成），所以我们可以通过查看它的源代码，来看它是如何实现的。

**③ ReenTrantLock 比 synchronized 增加了一些高级功能**

相比synchronized，ReenTrantLock增加了一些高级功能。主要来说主要有三点：**①等待可中断；②可实现公平锁；**

**③可实现选择性通知（锁可以绑定多个条件）ReenTrantLock提供了一种能够中断等待锁的线程的机制**，通过lock.lockInterruptibly()来实现这个机制。也就是说正在等待的线程可以选择放弃等待，改为处理其他事情。

**ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。** ReenTrantLock默认情况是非公平的，可以通过 ReenTrantLock类的ReentrantLock(boolean fair) 构造方法来制定是否是公平的。synchronized关键字与wait()和notify/notifyAll()方法相结合可以实现等待/通知机制，ReentrantLock类当然也 可以实现，但是需要借助于Condition接口与newCondition() 方法。Condition是JDK1.5之后才有的，它具有很

好的灵活性，比如可以实现多路通知功能也就是在一个Lock对象中可以创建多个Condition实例（即对象监视器），**线程对象可以注册在指定的Condition中，从而可以有选择性的进行线程通知，在调度线程上更加灵活。 在使用notify/notifyAll()方法进行通知时，被通知的线程是由 JVM 选择的，用ReentrantLock类结合Condition实例可以实现“选择性通知”** ，这个功能非常重要，而且是Condition接口默认提供的。而synchronized关键字就相当于整个Lock对象中只有一个Condition实例，所有的线程都注册在它一个身上。如果执行notifyAll()方法的话就会通知所有处于等待状态的线程这样会造成很大的效率问题，而Condition实例的signalAll()方法 只会唤醒注册在该Condition实例中的所有等待线程。如果你想使用上述功能，那么选择ReenTrantLock是一个不错的选择。

**④ 性能已不是选择标准**

### 3.2 面试中关于线程池的 4 连击

3.2.1 讲一下Java内存模型

在 JDK1.2 之前，Java的内存模型实现总是从**主存**（即共享内存）读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存**本地内存**（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成**数据的不一致**。

要解决这个问题，就需要把变量声明为 **volatile**，这就指示 JVM，这个变量是不稳定的，每次使用它都到主存中进行读取。说白了， **volatile** 关键字的主要作用就是保证变量的可见性然后还有一个作用是防止指令重排序。

**3.2.2 说说 synchronized 关键字和 volatile 关键字的区别**

synchronized关键字和volatile关键字比较

**volatile关键字**是线程同步的**轻量级实现**，所以**volatile性能肯定比synchronized关键字要好**。但是**volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块**。synchronized关键字在JavaSE1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，**实际开发中使用 synchronized 关键字的场景还是更多一些**。

**多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞**

**volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。**

**volatile关键字主要用于解决变量在多个线程之间的可见性，而 synchronized关键字解决的是多个线程之间访问资源的同步性。**

### 3.3面试中关于 线程池的 2 连击

**3.3.1 为什么要用线程池？**

线程池提供了一种限制和管理资源（包括执行一个任务）。 每个线程池还维护一些基本统计信息，例如已完成任务的数量。

**降低资源消耗。** 通过重复利用已创建的线程降低线程创建和销毁造成的消耗。

**提高响应速度。** 当任务到达时，任务可以不需要的等到线程创建就能立即执行。

**提高线程的可管理性。** 线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

**3.3.2 实现Runnable接口和Callable接口的区别**

如果想让线程池执行任务的话需要实现的Runnable接口或Callable接口。 Runnable接口或Callable接口实现类都可以被ThreadPoolExecutor或ScheduledThreadPoolExecutor执行。两者的区别在于 Runnable 接口不会返回结果但是 Callable 接口可以返回结果。

**备注：** 工具类 Executors 可以实现 Runnable 对象和 Callable 对象之间的相互转换。

（ Executors.callable（Runnable task） 或 Executors.callable（Runnable task，Object resule） ）。

**3.3.3 执行execute()方法和submit()方法的区别是什么呢？**

1) execute() **方法用于提交不需要返回值的任务，所以无法判断任务是否被线程池执行成功与否；**2)**submit()方法用于提交需要返回值的任务。线程池会返回一个future类型的对象，通过这个future对象可以判断任务是否执行成功**，并且可以通过future的get()方法来获取返回值，get()方法会阻塞当前线程直到任务完成，而使用get（long timeout，TimeUnit unit） 方法则会阻塞当前线程一段时间后立即返回，这时候有可能任务没有执行完。

**3.3.4 如何创建线程池**

《阿里巴巴Java开发手册》中强制线程池不允许使用 Executors 去创建，而是通过 ThreadPoolExecutor 的方式，这样的处理方式让写的同学更加明确线程池的运行规则，规避资源耗尽的风险

Executors 返回线程池对象的弊端如下：

**FixedThreadPool 和 SingleThreadExecutor** ： 允许请求的队列长度为 Integer.MAX\_VALUE,可能堆积

大量的请求，从而导致OOM。

**CachedThreadPool 和 ScheduledThreadPool** ： 允许创建的线程数量为 Integer.MAX\_VALUE ，可能

会创建大量线程，从而导致OOM。

**方式一：通过构造方法实现**

**方式二：通过Executor 框架的工具类Executors来实现** 我们可以创建三种类型的ThreadPoolExecutor：

**FixedThreadPool** ： 该方法返回一个固定线程数量的线程池。该线程池中的线程数量始终不变。当有一个新的任务提交时，线程池中若有空闲线程，则立即执行。若没有，则新的任务会被暂存在一个任务队列中，待有线程空闲时，便处理在任务队列中的任务。

**SingleThreadExecutor：** 方法返回一个只有一个线程的线程池。若多余一个任务被提交到该线程池，任务会被保存在一个任务队列中，待线程空闲，按先入先出的顺序执行队列中的任务。

**CachedThreadPool：** 该方法返回一个可根据实际情况调整线程数量的线程池。线程池的线程数量不确定，但若有空闲线程可以复用，则会优先使用可复用的线程。若所有线程均在工作，又有新的任务提交，则会创建新的线程处理任务。所有线程在当前任务执行完毕后，将返回线程池进行复用。

对应Executors工具类中的方法如图所示：

### 3.4 面试中关于 Atomic 原子类的 4 连击

**3.4.1 介绍一下Atomic 原子类**

Atomic 翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。在我们这里 Atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰。所以，所谓原子类说简单点就是具有原子/原子操作特征的类。并发包 java.util.concurrent 的原子类都存放在 java.util.concurrent.atomic 下,如下图所示。

**3.4.2 JUC 包中的原子类是哪4类?**

**基本类型**

使用原子的方式更新基本类型

AtomicInteger：整形原子类

AtomicLong：长整型原子类

AtomicBoolean ：布尔型原子类

**数组类型**

使用原子的方式更新数组里的某个元素

AtomicIntegerArray：整形数组原子类

AtomicLongArray：长整形数组原子类

AtomicReferenceArray ：引用类型数组原子类

**引用类型**

AtomicReference：引用类型原子类

AtomicStampedRerence：原子更新引用类型里的字段原子类

AtomicMarkableReference ：原子更新带有标记位的引用类型

**对象的属性修改类型**

AtomicIntegerFieldUpdater:原子更新整形字段的更新器AtomicLongFieldUpdater：原子更新长整形字段的更新器

AtomicStampedReference ：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决原

子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

**3.4.3 讲讲 AtomicInteger 的使用**

**AtomicInteger 类常用方法**

**AtomicInteger 类的使用示例**

使用 AtomicInteger 之后，不用对 increment() 方法加锁也可以保证线程安全。

**3.4.4 能不能给我简单介绍一下 AtomicInteger 类的原理**

AtomicInteger 线程安全原理简单分析

AtomicInteger 类的部分源码：

public final int get() //获取当前的值

public final int getAndSet(int newValue)//获取当前的值，并设置新的值

public final int getAndIncrement()//获取当前的值，并自增

public final int getAndDecrement() //获取当前的值，并自减

public final int getAndAdd(int delta) //获取当前的值，并加上预期的值

boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式将该值设置为输

入值（update）

public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可能导致其他线程

在之后的一小段时间内还是可以读到旧的值。

class AtomicIntegerTest {

private AtomicInteger count = new AtomicInteger();

//使用AtomicInteger之后，不需要对该方法加锁，也可以实现线程安全。

public void increment() {

count.incrementAndGet();

}

public int getCount() {

return count.get();

}

}AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免synchronized 的高开销，执行效率大为提升。

CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffffset。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

### 3.5 AQS

**3.5.1 AQS 介绍**

AQS的全称为（AbstractQueuedSynchronizer），这个类在java.util.concurrent.locks包下面。

AQS是一个用来构建锁和同步器的框架，使用AQS能简单且高效地构造出应用广泛的大量的同步器，比如我们提到的 ReentrantLock，Semaphore，其他的诸如ReentrantReadWriteLock，SynchronousQueue，FutureTask等等皆是

基于AQS的。当然，我们自己也能利用AQS非常轻松容易地构造出符合我们自己需求的同步器。

**3.5.2 AQS 原理分析**

// setup to use Unsafe.compareAndSwapInt for updates（更新操作时提供“比较并替换”的作用）

private static final Unsafe unsafe = Unsafe.getUnsafe();

private static final long valueOffset;

static {

try {

valueOffset = unsafe.objectFieldOffset

(AtomicInteger.class.getDeclaredField("value"));

} catch (Exception ex) { throw new Error(ex); }

}

private volatile int value;AQS 原理这部分参考了部分博客，在5.2节末尾放了链接。

在面试中被问到并发知识的时候，大多都会被问到“请你说一下自己对于AQS原理的理解”。下面给大家一个示例供大家参加，面试不是背题，大家一定要假如自己的思想，即使加入不了自己的思想也要保证自己能够通俗的讲出来而不是背出来。

下面大部分内容其实在AQS类注释上已经给出了，不过是英语看着比较吃力一点，感兴趣的话可以看看源码。

**3.5.3 AQS 原理概览**

**AQS核心思想是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并且将共享资源设置为锁定状态。如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。**

CLH(Craig,Landin,and Hagersten)队列是一个虚拟的双向队列（虚拟的双向队列即不存在队列实例，仅存在结点之间的关联关系）。AQS是将每条请求共享资源的线程封装成一个CLH锁队列的一个结点（Node）来实现锁

的分配。看个AQS(AbstractQueuedSynchronizer)原理图：

AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。AQS使用CAS对该同步状态进行原子操作实现对其值的修改。

private volatile int state;//共享变量，使用volatile修饰保证线程可见性

状态信息通过procted类型的getState，setState，compareAndSetState进行操作//返回同步状态的当前值

protected final int getState() {

return state;

}

// 设置同步状态的值

protected final void setState(int newState) {

state = newState;

}

//原子地（CAS操作）将同步状态值设置为给定值update如果当前同步状态的值等于expect（期望值）

protected final boolean compareAndSetState(int expect, int update) {

return unsafe.compareAndSwapInt(this, stateOffset, expect, update);

}

**3.5.3 AQS 对资源的共享方式**

**AQS定义两种资源共享方式**

**Exclusive**（独占）：只有一个线程能执行，如ReentrantLock。又可分为公平锁和非公平锁：

公平锁：按照线程在队列中的排队顺序，先到者先拿到锁

非公平锁：当线程要获取锁时，无视队列顺序直接去抢锁，谁抢到就是谁的

**Share**（共享）：多个线程可同时执行，如Semaphore/CountDownLatch。Semaphore、

CountDownLatCh、 CyclicBarrier、ReadWriteLock 我们都会在后面讲到。

ReentrantReadWriteLock 可以看成是组合式，因为ReentrantReadWriteLock也就是读写锁允许多个线程同时对某一资源进行读。不同的自定义同步器争用共享资源的方式也不同。自定义同步器在实现时只需要实现共享资源 state 的获取与释放方式即可，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。

**3.5.4 AQS底层使用了模板方法模式**

同步器的设计是基于模板方法模式的，如果需要自定义同步器一般的方式是这样（模板方法模式很经典的一个应用）：

1. 使用者继承AbstractQueuedSynchronizer并重写指定的方法。（这些重写方法很简单，无非是对于共享资源state的获取和释放）

2. 将AQS组合在自定义同步组件的实现中，并调用其模板方法，而这些模板方法会调用使用者重写的方法。这和我们以往通过实现接口的方式有很大区别，这是模板方法模式很经典的一个运用。

**AQS使用了模板方法模式，自定义同步器时需要重写下面几个AQS提供的模板方法：**

isHeldExclusively()//该线程是否正在独占资源。只有用到condition才需要去实现它。

tryAcquire(int)//独占方式。尝试获取资源，成功则返回true，失败则返回false。

tryRelease(int)//独占方式。尝试释放资源，成功则返回true，失败则返回false。

tryAcquireShared(int)//共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成

功，且有剩余资源。

tryReleaseShared(int)//共享方式。尝试释放资源，成功则返回true，失败则返回false。

默认情况下，每个方法都抛出 UnsupportedOperationException 。 这些方法的实现必须是内部线程安全的，并且通常应该简短而不是阻塞。AQS类中的其他方法都是fifinal ，所以无法被其他类使用，只有这几个方法可以被其他类使用。以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但

要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS(Compare and Swap)减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现 tryAcquire-tryRelease 、

tryAcquireShared-tryReleaseShared 中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，

如 ReentrantReadWriteLock 。

推荐两篇 AQS 原理和相关源码分析的文章：

http://www.cnblogs.com/waterystone/p/4920797.html

https://www.cnblogs.com/chengxiao/archive/2017/07/24/7141160.html

5.3 AQS 组件总结

**Semaphore(信号量)-允许多个线程同时访问：** synchronized 和 ReentrantLock 都是一次只允许一个线程访问

某个资源，Semaphore(信号量)可以指定多个线程同时访问某个资源。

**CountDownLatch （倒计时器）：** CountDownLatch是一个同步工具类，用来协调多个线程之间的同步。这

个工具通常用来控制线程等待，它可以让某一个线程等待直到倒计时结束，再开始执行。

**CyclicBarrier(循环栅栏)：** CyclicBarrier 和 CountDownLatch 非常类似，它也可以实现线程间的技术等待，但是它的功能比 CountDownLatch 更加复杂和强大。主要应用场景和 CountDownLatch 类似。CyclicBarrier 的字面意思是可循环使用（Cyclic）的屏障（Barrier）。它要做的事情是，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续干活。

CyclicBarrier默认的构造方法是 CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。

## 4.工作问题

### 4.1如果面试官问你“你有什么问题问我吗？”时，你该如何回答

针对这个问题。笔主专门找了几个专门做HR工作的小哥哥小姐姐们询问并且查阅了挺多前辈们的回答，然后结合自己的实际经历，我概括了下面几个比较适合问的问题。

**4.1.1面对HR或者其他Level比较低的面试官时**

1. **能不能谈谈你作为一个公司老员工对公司的感受?** (这个问题比较容易回答，不会让面试官陷入无话可说的尴尬境地。另外，从面试官的回答中你可以加深对这个公司的了解，让你更加清楚这个公司到底是不是你想的那样或者说你是否能适应这个公司的文化。除此之外，这样的问题在某种程度上还可以拉进你与面试官的距离。)

2. **能不能问一下，你当时因为什么原因选择加入这家公司的呢或者说这家公司有哪些地方吸引你?有什么地方你觉得还不太好或者可以继续完善吗？** （类似第一个问题，都是问面试官个人对于公司的看法，）

3. **我觉得我这次表现的不是太好，你有什么建议或者评价给我吗？**(这个是我常问的。我觉得说自己表现不好只是这个语境需要这样来说，这样可以显的你比较谦虚好学上进。)

4. **接下来我会有一段空档期，有什么值得注意或者建议学习的吗？** （体现出你对工作比较上心，自助学习意识比较强。）

5. **这个岗位为什么还在招人？** (岗位真实性和价值咨询)

6. **大概什么时候能给我回复呢？** (终面的时候，如果面试官没有说的话，可以问一下)

7. ......

4.1.2面对部门领导

1. **部门的主要人员分配以及对应的主要工作能简单介绍一下吗？**

2. **未来如果我要加入这个团队，你对我的期望是什么？** （部门领导一般情况下是你的直属上级了，你以后和他打交道的机会应该是最多的。你问这个问题，会让他感觉你是一个对他的部门比较上心，比较有团体意识，并且愿意倾听的候选人。）

3. **公司对新入职的员工的培养机制是什么样的呢？** （正规的公司一般都有培养机制，提前问一下是对你自己的负责也会显的你比较上心）

4. **以您来看，这个岗位未来在公司内部的发展如何？** (在我看来，问这个问题也是对你自己的负责吧，谁不想发展前景更好的岗位呢？)

5. **团队现在面临的最大挑战是什么？** (这样的问题不会暴露你对公司的不了解，并且也能让你对未来工作的挑战或困难有一个提前的预期。)

**4.1.3面对Level比较高的(比如总裁,老板)**

1. **贵公司的发展目标和方向是什么？** （看下公司的发展是否满足自己的期望）

2. **与同行业的竞争者相比，贵公司的核心竞争优势在什么地方？** （充分了解自己的优势和劣势）

3. **公司现在面临的最大挑战是什么？**

### 4.2 面试官问你的优点是什么，应该如何回答?

回答这样的问题，最好能够结合你要应聘的职位来做针对性回答。一般面试官问这个问题的时候，很可能会只让你说几个你觉得最能体现你能力的优点，为了避免自己在面试过程中不知道该说自己的那些优点，你可以在面试之前好好准备一下。 面试的时候最好可以说几个你要应聘的职位所做的事情需要的优点或者说你要应聘的公司比较看重的优点(企业文化)。

### 4.3 面试官问你的缺点是什么，应该如何回答?

缺点肯定不能是目标岗位需要的关键能力！！！

总之，记住一点，面试官问你这个问题的话，你可以说一些不影响你这个职位工作需要的一些缺点。比如你面试后端 工程师，面试官问你的缺点是什么的话，你可以这样说：自己比较内向，平时不太爱与人交流，但是考虑到以后可能要和客户沟通，自己正在努力改。

### 4.4 项目中使用到的应用服务器是什么？

Tomcat+nginx

### 4.5 讲一下每台服务器的集群数量：

项目中一共 14 台项目服务，那么为了每一台高可用一主一备，但首页项目高并发设为

四台服务器，则一共 32 台项目服务器，再加 redis 集群用了 3 台，为了每一台高可用一主一 备一共 6 台，es集群 7 台 服务器，nginx 为了高可用一主一备一共 2 台，mysql 数据库集群 3 台！Rabbitmq 消息中间件高可用 2 台；

共计：58 台服务器！

**4.6你在项目开发中碰到过哪些重大且棘手的问题**

场景一：需求不明确困境

在项目开发中，项目采用迭代开发，开发需求不是很明确，对于项目开发初期来说非常

困难，进度非常慢，有时开发的出的产品结果往往不能令老板满意，或者是甲方满意，项目

还需要不停的迭代，修改。比如说：在开发品优购项目的时候，客户定位是一个综合性的商

务平台，可以实现在线第三方商家对接，实现商品的销售但是并没有明确的需求，因此开发

全凭借电商的项目经验来实现里面的相关的业务，后期慢慢迭代。

场景二：使用 cas 单独登录不能很好实现想要的效果

场景三：Nginx 静态页面服务跳转到购物车跨域问题

在 Nginx 中部署了静态页面，添加购物车时必须从静态页面跳转到购物车系统,实现购物

车添加操作。

由于在静态页面中使用 angularJS 实现的跳转，发现跳转到购物车系统完全没有问题，但是

并不能跳转回到购物车系统页面。

问题分析：

从静态详情系统跳转到购物车系统，会存在跨域问题，因此不能进行回调函数的数据传递。

### 4.6 项目用到的版本号

根据项目时间,去官网上找项目时间前几年的版本号.

# Java Web

## 1.JDBC 技术

### 1.1 说下原生 jdbc 操作数据库流程？

第一步：Class.forName()加载数据库连接驱动；

第二步：DriverManager.getConnection()获取数据连接对象;

第三步：根据 SQL 获取 sql 会话对象，有 2 种方式 Statement、PreparedStatement ;

第四步：执行 SQL 处理结果集，执行 SQL 前如果有参数值就设置参数值 setXXX();

第五步：关闭结果集、关闭会话、关闭连接。

### 1.2说说事务的概念，在JDBC编程中处理事务的步骤。 　　1 事务是作为单个逻辑工作单元执行的一系列操作。 　　2，一个逻辑工作单元必须有四个属性，称为原子性、一致性、隔离性和持久性 (ACID) 属性，只有这样才能成为一个事务事务处理步骤： 　　3，conn.setAutoComit(false);设置提交方式为手工提交 　　4，conn.commit()提交事务 　　5，出现异常，回滚 conn.rollback();

### **1.3JDBC的脏读是什么？哪种数据库隔离级别能防止脏读？**

　　当我们使用事务时，有可能会出现这样的情况，有一行数据刚更新，与此同时另一个查询读到了这个刚更新的值。这样就导致了脏读，因为更新的数据还没有进行持久化，更新这行数据的业务可能会进行回滚，这样这个数据就是无效的。数据库的 TRANSACTIONREADCOMMITTED ，TRANSACTIONREPEATABLEREAD 和 TRANSACTION\_SERIALIZABLE 隔离级别可以防止脏读

## 2.网路通讯部分

### 2.1TCP与UDP区别？

UDP:

a、是面向无连接, 将数据及源的封装成数据包中,不需要建立建立连接

      b、每个数据报的大小在限制64k内

      c、因无连接,是不可靠协议

    d、不需要建立连接,速度快

TCP：

a、建议连接，形成传输数据的通道.

    b、在连接中进行大数据量传输，以字节流方式

      C、通过三次握手完成连接,是可靠协议

      d、必须建立连接m效率会稍低

      聊天、网络视频会议、桌面共享用的就是UDP

### 2.2说一下什么是Http协议？

客户端和 服务器端之间数据传输的格式规范，格式简称为“超文本传输协议”。 是一个基于请求与响应模式的、无状态的、应用层的协议，基于TCP的连接方式

### 2.3get与post请求区别？

区别一：

  get重点在从服务器上获取资源，post重点在向服务器发送数据；

区别二：

   get传输数据是通过URL请求，以field（字段）= value的形式，置于URL后，并用"?"连接，多个请求数据间用"&"连接，如http://127.0.0.1/Test/login.action?name=admin&password=admin，这个过程用户是可见的；

post传输数据通过Http的post机制，将字段与对应值封存在请求实体中发送给服务器，这个过程对用户是不可见的；

区别三：

    Get传输的数据量小，因为受URL长度限制，但效率较高；

Post可以传输大量数据，所以上传文件时只能用Post方式；

区别四：

   get是不安全的，因为URL是可见的，可能会泄露私密信息，如密码等；

post较get安全性较高；

区别五：

    get方式只能支持ASCII字符，向服务器传的中文字符可能会乱码。

post支持标准字符集，可以正确传递中文字符。

### 2.4 http 中重定向和请求转发的区别？

本质区别：转发是服务器行为，重定向是客户端行为。

重定向特点：两次请求，浏览器地址发生变化，可以访问自己 web 之外的资源，传输的数据会丢失。

请求转发特点：一次强求，浏览器地址不变，访问的是自己本身的 web 资源，传输的数据不会丢失。

## 3.Cookie 和 和 Session

### 3.1 Cookie 和 Session 的区别

**Cookie** 是 web 服务器发送给浏览器的一块信息，浏览器会在本地一个文件中给每个 web 服务器存储 cookie。以后浏览器再给特定的 web 服务器发送请求时，同时会发送所有为该服务器存储的 cookie。

**Session** 是存储在 web 服务器端的一块信息。session 对象存储特定用户会话所需的属性及配置信息。当用户在应用程序的 Web 页之间跳转时，存储在 Session 对象中的变量将不会丢失，而是在整个用户会话中一直存在下去。

Cookie 和 session 的不同点：

1、无论客户端做怎样的设置，session 都能够正常工作。当客户端禁用 cookie 时将无法使用 cookie。

2、在存储的数据量方面：session 能够存储任意的 java 对象，cookie 只能存储 String 类型的对象。

## 4.Jsp和Servlet

### 4.1.讲下Servlet的执行流程。

Servlet的执行流程也就是servlet的生命周期，当服务器启动的时候生命周期开始，然后通过init()《启动顺序根据web.xml里的startup-on-load来确定加载顺序》方法初始化servlet，再根据不同请求调用doGet或doPost方法，最后再通过destroy()方法进行销毁。

### 4.2.doGet和doPost的区别

doGet和doPost都是接受用户请求的方法，doGet处理get请求，doPost处理post请求，doGet用于地址栏提交，doPost用于表单提交，在页面提交数据时，get的数据大小有限制4k，post没有限制，get请求提交的数据会在地址栏显示，post不显示，所以post比get安全.

### 4.3.Jsp和servlet的区别

jsp的可读性强，容易维护，并且jsp在最后会编译成servlet

servlet容易调试

### 4.4.JSP九大内置对象:

pageContext ：只对当前jsp页面有效，里面封装了基本的request和session的对象  
 Request ：对当前请求进行封装  
 Session ：浏览器会话对象，浏览器范围内有效  
 Application ：应用程序对象，对整个web工程都有效  
 Out ：页面打印对象，在jsp页面打印字符串  
 Response ：返回服务器端信息给用户  
 Config ：单个servlet的配置对象，相当于servletConfig对象  
 Page ：当前页面对象，也就是this  
 Exception ：错误页面的exception对象，如果指定的是错误页面，这个就是异常对象

### 4.5.三大指令：

Page ：指令是针对当前页面的指令  
 Include ：用于指定如何包含另一个页面  
 Taglib ：用于定义和指定自定义标签

### 4.6七大动作：

Forward，执行页面跳转，将请求的处理转发到另一个页面  
 Param ：用于传递参数  
 Include ：用于动态引入一个jsp页面  
 Plugin ：用于下载javaBean或applet到客户端执行  
 useBean ：使用javaBean  
 setProperty ：修改javaBean实例的属性值  
 getProperty ：获取javaBean实例的属性值

## 5.Jquery、ajax

### 5.1. 谈谈你对 ajax 的认识？

Ajax 是一种创建交互式网页应用的的网页开发技术；Asynchronous JavaScript and XML”的缩写。

Ajax 的优势：

通过异步模式，提升了用户体验。

优化了浏览器和服务器之间的传输，减少不必要的数据往返，减少了带宽占用。

Ajax 引擎在客户端运行，承担了一部分本来由服务器承担的工作，从而减少了大用户量下的服务器负载。

Ajax 的最大特点：

可以实现局部刷新，在不更新整个页面的前提下维护数据，提升用户体验度。

### 5.2.手写ajax

$.ajax({

url:'http://www.baidu.com',

type:'POST',

data:data,

cache:true,

headers:{},

beforeSend：function(){},

success:function(){},

error:function(){},

complete:function(){}

});

### 5.3.Ajax你以前用过么?简单介绍一下

AJAX = 异步 JavaScript 和 XML。

AJAX 是一种用于创建快速动态网页的技术。

通过在后台与服务器进行少量数据交换，AJAX 可以使网页实现异步更新。这意味着可以在不重新加载整个网页的情况下，对网页的某部分进行更新。

传统的网页（不使用 AJAX）如果需要更新内容，必需重载整个网页面。

有很多使用 AJAX 的应用程序案例：新浪微博、Google 地图、开心网等等。

### 5.4.Ajax可以做异步请求么?

可以.ajax请求默认是异步的.如果想同步 把 async设置为false就可以了默认是true

如果是jquery

 $.ajax({

url: some.php,

async: false,

success : function(){

}

});

如果是原生的js

xmlHttp.open("POST",url,false);

### 5.5. jsonp 原理

jsonp 的最基本的原理是：动态添加一个<script>标签，使用 script 标签的 src 属性没有跨域的限制的特点实现跨域。首先在客户端注册一个 callback, 然后把 callback 的名字传给服务器。此时，服务器先生成 json 数据。 然后以 javascript 语法的方式，生成一个 function , function 名字就是传递上来的参数 jsonp。最后将json 数据直接以入参的方式，放置到 function 中，这样就生成了一段 js 语法的文档，返回给客户端。客户端浏览器，解析 script 标签，并执行返回的 javascript 文档，此时数据作为参数，传入到了客户端预先定义好的 callback 函数里。

### 5.6.Jquery常用选择器

1、ID选择器 #id

描述：根据给定的id匹配一个元素， 返回单个元素（注：在网页中，id名称不能重复）

示例：$("#test") 选取 id 为 test 的元素

2、类选择器 .class

描述：根据给定的类名匹配元素，返回元素集合

示例：$(".test") 选取所有class为test的元素

3、元素选择器 element

描述：根据给定的元素名匹配元素，返回元素集合

示例：$("p") 选取所有的<p>元素

4、\*

描述：匹配所有元素，返回元素集合

示例：$("\*") 选取所有的元素

## 6.Angular中是怎么实现数据双向绑定的？

Angular实现了双向绑定机制。所谓的双向绑定，无非是从界面的操作能实时反映到数据，数据的变更能实时展现到界面。即数据模型（Module）和视图（View）之间的双向绑定。

脏检查机制。双向数据绑定是 AngularJS 的核心机制之一。当 view 中有任何数据变化时，会更新到 model ，当 model 中数据有变化时，view 也会同步更新，显然，这需要一个监控。

原理就是，Angular 在 scope 模型上设置了一个 监听队列，用来监听数据变化并更新 view 。每次绑定一个东西到 view 上时 AngularJS 就会往 $watch 队列里插入一条 $watch，用来检测它监视的 model 里是否有变化的东西。当浏览器接收到可以被 angular context 处理的事件时，$digest 循环就会触发，遍历所有的 $watch，最后更新 dom。

# 三.数据库

## 3.1 Sql 之连接查询

外连接：

1） 左连接（左外连接）以左表为基准进行查询,左表数据会全部显示出来,右表 如果和左表匹配 的数据则显示相应字段的数据,如果不匹配,则显示为 NULL;

2） 右连接（右外连接）以右表为基准进行查询,右表数据会全部显示出来,右表 如果和左表匹配的数据则显示相应字段的数据,如果不匹配,则显示为 NULL;

3） 全连接就是先以左表进行左外连接，然后以右表进行右外连接。

内连接：

显示表之间有连接匹配的所有行。

## 3.2 Sql 之聚合函数：

聚合函数是对一组值执行计算并返回单一的值的函数，它经常与 SELECT 语句的 GROUP BY 子句一同使用。

1).AVG 返回指定组中的平均值，空值被忽略； COUNT 返回指定组中 项目的数量。 例：select prd\_no,avg(qty) from sales group by prd\_no

2). MAX 返回指定数据的最大值；MIN 返回指定数据的最小值；SUM 返回指定数据的和，只能用于数字列，空值被忽略。 例：select prd\_no,max(qty) from sales group by prd\_no

3）使用 group by 子句对数据进行分组；对 group by 子句形成的组运行 聚集函数计算每一组的值；最后用 having 子句去掉不符合条件的组；having 子 句中的每一个元素也必须出现在 select 列表中。有些数据库例外，如 oracle. 例：select prd\_no,max(qty) from sales group by prd\_no having prd\_no>10

## 3.3 Sql 之 SQL 注入

举例：

select admin from user where username='admin' or 'a'='a' and passwd=''or 'a'='a'

防止 SQL 注入，使用预编译语句是预防 SQL 注入的最佳方式，如

select admin from user where username=？And password=?

使用预编译的 SQL 语句语义不会发生改变，在 SQL 语句中，变量用问号？ 表示。像上面例子中，username 变量传递的'admin' or 'a'='a' 参数，也只会当 作 username 字符串来解释查询，从根本上杜绝了 SQL 注入攻击的发生。

注意：使用 mybaits 时 mapper 中#方式能够很大程度防止 sql 注入，$方式 无法防止 sql 注入.

## 3.4 SQL Select 语句完整的执行顺序:

from--->where--->group by--->having--->计算所有的表达式--->order by-- ->select 输出

## 3.5存储引擎

1. 概念

数据库存储引擎是数据库底层软件组织，数据库管理系统（DBMS）使用数据引擎进行创建、查询、 更新和删除数据。不同的存储引擎提供不同的存储机制、索引技巧、锁定水平等功能，使用不同 的存储引擎，还可以 获得特定的功能。现在许多不同的数据库管理系统都支持多种不同的数据引 擎。

存储引擎主要有： 1. MyIsam , 2. InnoDB, 3. Memory, 4. Archive, 5. Federated 。

2. InnoDB

InnoDB 底层存储结构为B+树， B树的每个节点对应innodb的一个page，page大小是固定的， 一般设为 16k。其中非叶子节点只有键值，叶子节点包含完成数据。

适用场景：

1）经常更新的表，适合处理多重并发的更新请求。

2）支持事务。

3）可以从灾难中恢复（通过 bin-log 日志等）。

4）外键约束。只有他支持外键。

5）支持自动增加列属性 auto\_increment。

3. TokuDB

TokuDB 底层存储结构为 Fractal Tree,Fractal Tree 的结构与 B+树有些类似, 在 Fractal Tree 中，每一个 child 指针除了需要指向一个 child 节点外，还会带有一个 Message Buffer ，这个 Message Buffer 是一个 FIFO 的队列，用来缓存更新操作。

例如，一次插入操作只需要落在某节点的 Message Buffer 就可以马上返回了，并不需要搜索到叶 子节点。这些缓存的更新会在查询时或后台异步合并应用到对应的节点中。

TokuDB 在线添加索引，不影响读写操作, 非常快的写入性能， Fractal-tree 在事务实现上有优 势。 他主要适用于访问频率不高的数据或历史数据归档。

4. MyIASM

MyIASM是 MySQL默认的引擎，但是它没有提供对数据库事务的支持，也不支持行级锁和外键， 因此当 INSERT(插入)或 UPDATE(更新)数据时即写操作需要锁定整个表，效率便会低一些。

ISAM 执行读取操作的速度很快，而且不占用大量的内存和存储资源。在设计之初就预想数据组织 成有固定长度的记录，按顺序存储的。---ISAM 是一种静态索引结构。 缺点是它不 支持事务处理。

5. Memory

Memory（也叫 HEAP）堆内存：使用存在内存中的内容来创建表。每个 MEMORY 表只实际对应 一个磁盘文件。MEMORY 类型的表访问非常得快，因为它的数据是放在内存中的，并且默认使用 HASH 索引。但是一旦服务关闭，表中的数据就会丢失掉。 Memory 同时支持散列索引和 B 树索 引，B树索引可以使用部分查询和通配查询，也可以使用和>=等操作符方便数据挖掘，散列索 引相等的比较快但是对于范围的比较慢很多。

## 3.6 索引

索引（Index）是帮助 MySQL 高效获取数据的数据结构。常见的查询算法,顺序查找,二分查找,二 叉排序树查找,哈希散列法,分块查找,平衡多路搜索树 B 树（B-tree）

索引就是加快检索表中数据的方法。数据库的索引类似于书籍的索引。在书籍中，索引允许用户不必翻阅完整个书就能迅速地找到所需要的信息。在数据库中，索引也允许数据库程序迅速地找到表中的数据，而不必扫描整个数据库。

MySQL数据库几个基本的索引类型：普通索引、唯一索引、主键索引、全文索引、组合索引

1.普通索引

是最基本的索引，它没有任何限制。它有以下几种创建方式：

（1）直接创建索引

|  |  |
| --- | --- |
| 1 | CREATE INDEX index\_name ON table(column(length)) |

（2）修改表结构的方式添加索引

|  |  |
| --- | --- |
| 1 | ALTER TABLE table\_name ADD INDEX index\_name ON (column(length)) |

（3）创建表的时候同时创建索引

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE `table` (      `id` int(11) NOT NULL AUTO\_INCREMENT ,      `title` char(255) CHARACTER NOT NULL ,      `content` text CHARACTER NULL ,      `time` int(10) NULL DEFAULT NULL ,      PRIMARY KEY (`id`),      INDEX index\_name (title(length))  ) |

（4）删除索引

|  |  |
| --- | --- |
| 1 | DROP INDEX index\_name ON table |

2.唯一索引

与前面的普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值。如果是组合索引，则列值的组合必须唯一。它有以下几种创建方式：

（1）创建唯一索引

|  |  |
| --- | --- |
| 1 | CREATE UNIQUE INDEX indexName ON table(column(length)) |

（2）修改表结构

|  |  |
| --- | --- |
| 1 | ALTER TABLE table\_name ADD UNIQUE indexName ON (column(length)) |

（3）创建表的时候直接指定

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | CREATE TABLE `table` (      `id` int(11) NOT NULL AUTO\_INCREMENT ,      `title` char(255) CHARACTER NOT NULL ,      `content` text CHARACTER NULL ,      `time` int(10) NULL DEFAULT NULL ,      UNIQUE indexName (title(length))  ); |

3.主键索引

是一种特殊的唯一索引，一个表只能有一个主键，不允许有空值。一般是在建表的时候同时创建主键索引：

|  |  |
| --- | --- |
| 1  2  3  4  5 | CREATE TABLE `table` (      `id` int(11) NOT NULL AUTO\_INCREMENT ,      `title` char(255) NOT NULL ,      PRIMARY KEY (`id`)  ); |

4.组合索引

指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用组合索引时遵循最左前缀集合

|  |  |
| --- | --- |
| 1 | ALTER TABLE `table` ADD INDEX name\_city\_age (name,city,age); |

5.全文索引

主要用来查找文本中的关键字，而不是直接与索引中的值相比较。fulltext索引跟其它索引大不相同，它更像是一个搜索引擎，而不是简单的where语句的参数匹配。fulltext索引配合match against操作使用，而不是一般的where语句加like。它可以在create table，alter table ，create index使用，不过目前只有char、varchar，text 列上可以创建全文索引。值得一提的是，在数据量较大时候，现将数据放入一个没有全局索引的表中，然后再用CREATE index创建fulltext索引，要比先为一张表建立fulltext然后再将数据写入的速度快很多。

（1）创建表的适合添加全文索引

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE `table` (      `id` int(11) NOT NULL AUTO\_INCREMENT ,      `title` char(255) CHARACTER NOT NULL ,      `content` text CHARACTER NULL ,      `time` int(10) NULL DEFAULT NULL ,      PRIMARY KEY (`id`),      FULLTEXT (content)  ); |

（2）修改表结构添加全文索引

|  |  |
| --- | --- |
| 1 | ALTER TABLE article ADD FULLTEXT index\_content(content) |

（3）直接创建索引

|  |  |
| --- | --- |
| 1 | CREATE FULLTEXT INDEX index\_content ON article(content) |

索引的优点

创建唯一性索引，保证数据库表中每一行数据的唯一性

大大加快数据的检索速度，这也是创建索引的最主要的原因

加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

通过使用索引，可以在查询的过程中使用优化隐藏器，提高系统的性能。

索引的缺点

创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加

索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大

当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，降低了数据的维护速度

3.常见索引原则有 :

选择唯一性索引：唯一性索引的值是唯一的，可以更快速的通过该索引来确定某条记录。

为经常需要排序、分组和联合操作的字段建立索引.

为常作为查询条件的字段建立索引。

限制索引的数目：越多的索引，会使更新表变得很浪费时间。

尽量使用数据量少的索引：如果索引的值很长，那么查询的速度会受到影响。

尽量使用前缀来索引：如果索引字段的值很长，最好使用值的前缀来索引。

删除不再使用或者很少使用的索引

最左前缀匹配原则，非常重要的原则。

尽量选择区分度高的列作为索引：区分度的公式是表示字段不重复的比例

索引列不能参与计算，保持列“干净”：带函数的查询不参与索引。

尽量的扩展索引，不要新建索引。

## 3.7数据库三范式

范式是具有最小冗余的表结构。3 范式具体如下：

1.第一范式(1st NF －First Normal Fromate)

第一范式的目标是确保每列的原子性:如果每列都是不可再分的最小数据单元（也称为最小的原子 单元），则满足第一范式（1NF）

第一范式（1NF）要求数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值。

若某一列有多个值，可以将该列单独拆分成一个实体，新实体和原实体间是一对多的关系。

在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。

第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。

第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到“地址”这个属性，本来直接将“地址”属性设计成一个数据库表的字段就行。但是如果系统经常会访问“地址”属性中的“城市”部分，那么就非要将“地址”这个属性重新拆分为省份、城市、详细地址等多个部分进行存储，这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式

2.第二范式(2nd NF－Second Normal Fromate)

首先满足第一范式，并且表中非主键列不存在对主键的部分依赖。 第二范式要求每个表只描述一 件事情。

满足第二范式（2NF）必须先满足第一范式（1NF）。

第二范式要求实体中没一行的所有非主属性都必须完全依赖于主键；即：非主属性必须完全依赖于主键。

完全依赖：主键可能由多个属性构成，完全依赖要求不允许存在非主属性依赖于主键中的某一部分属性。

若存在哪个非主属性依赖于主键中的一部分属性，那么要将发生部分依赖的这一组属性单独新建一个实体，并且在旧实体中用外键与新实体关联，并且新实体与旧实体间是一对多的关系。

第二范式在第一范式的基础之上更进一层。第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。

3.第三范式(3rd NF－ Third Normal Fromate)

第三范式定义是，满足第二范式，并且表中的列不存在对非主键列的传递依赖。除了主键订单编 号外，顾客姓名依赖于非主键顾客编号。

满足第三范式必须先满足第二范式。

第三范式要求：实体中的属性不能是其他实体中的非主属性。因为这样会出现冗余。即：属性不依赖于其他非主属性。

如果一个实体中出现其他实体的非主属性，可以将这两个实体用外键关联，而不是将另一张表的非主属性直接写在当前表中。

第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

## 3.8. 数据库事务

1. 事务(TRANSACTION)是作为单个逻辑工作单元执行的一系列操作，这些操作作为一个整体一起向 系统提交，要么都执行、要么都不执行 。事务是一个不可分割的工作逻辑单元 事务必须具备以下四个属性，简称 ACID 属性：

1. 原子性（Atomicity）：事务是一个完整的操作。事务的各步操作是不可分的（原子的）；要 么都执行，要么都不执行。
2. 一致性（Consistency）：当事务完成时，数据必须处于一致状态。
3. 隔离性（Isolation）：对数据进行修改的所有并发事务是彼此隔离的，这表明事务必须是独 立的，它不应以任何方式依赖于或影响其他事务。
4. 永久性（Durability）：事务完成后，它对数据库的修改被永久保持，事务日志能够保持事务 的永久性。

2.事务控制语句：

* BEGIN 或 START TRANSACTION 显式地开启一个事务；
* COMMIT 也可以使用 COMMIT WORK，不过二者是等价的。COMMIT 会提交事务，并使已对数据库进行的所有修改成为永久性的；
* ROLLBACK 也可以使用 ROLLBACK WORK，不过二者是等价的。回滚会结束用户的事务，并撤销正在进行的所有未提交的修改；
* SAVEPOINT identifier，SAVEPOINT 允许在事务中创建一个保存点，一个事务中可以有多个 SAVEPOINT；
* RELEASE SAVEPOINT identifier 删除一个事务的保存点，当没有指定的保存点时，执行该语句会抛出一个异常；
* ROLLBACK TO identifier 把事务回滚到标记点；
* SET TRANSACTION 用来设置事务的隔离级别。InnoDB 存储引擎提供事务的隔离级别有READ UNCOMMITTED、READ COMMITTED、REPEATABLE READ 和 SERIALIZABLE。

3.MYSQL 事务处理主要有两种方法：

1. 用 BEGIN, ROLLBACK, COMMIT来实现
   1. BEGIN 开始一个事务
   2. ROLLBACK 事务回滚
   3. COMMIT 事务确认
2. 直接用 SET 来改变 MySQL 的自动提交模式:
   1. SET AUTOCOMMIT=0 禁止自动提交
   2. SET AUTOCOMMIT=1 开启自动提交

4. 事务的四种隔离级别

1. Read uncommitted

读未提交，顾名思义，就是一个事务可以读取另一个未提交事务的数据。

1. Read committed

读已提交，顾名思义，就是一个事务要等另一个事务提交后才能读取数据。

1. Repeatable read

可重复读，就是在开始读取数据（事务开启）时，不再允许修改操作

1. Serializable 序列化

Serializable 是最高的事务隔离级别，在该级别下，事务串行化顺序执行，可以避免脏读、不可重复读与幻读。但是这种事务隔离级别效率低下，比较耗数据库性能，一般不使用。

**在MySQL数据库中，支持上面四种隔离级别，默认的为Repeatable read (可重复读)**；而在Oracle数据库中，只支持Serializable (串行化)级别和Read committed (读已提交)这两种级别，其中默认的为Read committed级别。

## 3.9. 存储过程

一组为了完成特定功能的 SQL 语句集，存储在数据库中，经过第一次编译后再次调用不需要再次 编译，用户通过指定存储过程的名字并给出参数（如果该存储过程带有参数）来执行它。存储过 程是数据库中的一个重要对象。

存储过程优化思路：

1. 尽量利用一些 sql 语句来替代一些小循环，例如聚合函数，求平均函数等。

2.中间结果存放于临时表，加索引。

3.少使用游标。sql 是个集合语言，对于集合运算具有较高性能。而 cursors 是过程运算。比 如对一个 100 万行的数据进行查询。游标需要读表 100 万次，而不使用游标则只需要少量几 次读取。

4.事务越短越好。sqlserver支持并发操作。如果事务过多过长，或者隔离级别过高，都会造 成并发操作的阻塞，死锁。导致查询极慢，cpu 占用率极地。

5.使用 try-catch 处理错误异常。

6.查找语句尽量不要放在循环内。

## 3.10触发器

触发器是一段能自动执行的程序，是一种特殊的存储过程，触发器和普通的存储过程的区别是： 触发器是当对某一个表进行操作时触发。诸如：update、insert、delete 这些操作的时候，系统 会自动调用执行该表上对应的触发器。SQL Server 2005 中触发器可以分为两类：DML 触发器和 DDL 触发器，其中 DDL 触发器它们会影响多种数据定义语言语句而激发，这些语句有 create、 alter、drop 语句。

## 3.11 数据库并发策略

并发控制一般采用三种方法，分别是乐观锁和悲观锁以及时间戳。

乐观锁

乐观锁认为一个用户读数据的时候，别人不会去写自己所读的数据；悲观锁就刚好相反，觉得自 己读数据库的时候，别人可能刚好在写自己刚读的数据，其实就是持一种比较保守的态度；时间 戳就是不加锁，通过时间戳来控制并发出现的问题。

悲观锁

悲观锁就是在读取数据的时候，为了不让别人修改自己读取的数据，就会先对自己读取的数据加 锁，只有自己把数据读完了，才允许别人修改那部分数据，或者反过来说，就是自己修改某条数 据的时候，不允许别人读取该数据，只有等自己的整个事务提交了，才释放自己加上的锁，才允 许其他用户访问那部分数据。

两种锁的使用场景

从上面对两种锁的介绍，我们知道两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下（多读场景），即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果是多写的情况，一般会经常产生冲突，这就会导致上层应用会不断的进行retry，这样反倒是降低了性能，所以一般多写的场景下用悲观锁就比较合适。

乐观锁常见的两种实现方式

版本号机制

一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功。

CAS算法即

compare and swap（比较与交换），是一种有名的无锁算法。无锁编程，即不使用锁的情况下实现多线程之间的变量同步，也就是在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步（Non-blocking Synchronization）。CAS算法涉及到三个操作数

需要读写的内存值 V

进行比较的值 A

拟写入的新值 B

当且仅当 V 的值等于 A时，CAS通过原子方式用新值B来更新V的值，否则不会执行任何操作（比较和替换是一个原子操作）。一般情况下是一个自旋操作，即不断的重试。

乐观锁的缺点

ABA 问题

如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA"问题。

JDK 1.5 以后的 AtomicStampedReference 类就提供了此种能力，其中的 compareAndSet 方法就是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

循环时间长开销大

自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。 如果JVM能支持处理器提供的pause指令那么效率会有一定的提升，pause指令有两个作用，第一它可以延迟流水线执行指令（de-pipeline）,使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零。第二它可以避免在退出循环的时候因内存顺序冲突（memory order violation）而引起CPU流水线被清空（CPU pipeline flush），从而提高CPU的执行效率。

只能保证一个共享变量的原子操作

CAS 只对单个共享变量有效，当操作涉及跨多个共享变量时 CAS 无效。但是从 JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，你可以把多个变量放在一个对象里来进行 CAS 操作.所以我们可以使用锁或者利用AtomicReference类把多个共享变量合并成一个共享变量来操作

CAS与synchronized的使用情景

简单的来说CAS适用于写比较少的情况下（多读场景，冲突一般较少），synchronized适用于写比较多的情况下（多写场景，冲突一般较多）

对于资源竞争较少（线程冲突较轻）的情况，使用synchronized同步锁进行线程阻塞和唤醒切换以及用户态内核态间的切换操作额外浪费消耗cpu资源；而CAS基于硬件实现，不需要进入内核，不需要切换线程，操作自旋几率较少，因此可以获得更高的性能。

对于资源竞争严重（线程冲突严重）的情况，CAS自旋的概率会比较大，从而浪费更多的CPU资源，效率低于synchronized。

补充： Java并发编程这个领域中synchronized关键字一直都是元老级的角色，很久之前很多人都会称它为 “重量级锁” 。但是，在JavaSE 1.6之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的 偏向锁 和 轻量级锁 以及其它各种优化之后变得在某些情况下并不是那么重了。synchronized的底层实现主要依靠 Lock-Free 的队列，基本思路是 自旋后阻塞，竞争切换后继续竞争锁，稍微牺牲了公平性，但获得了高吞吐量。在线程冲突较少的情况下，可以获得和CAS类似的性能；而线程冲突严重的情况下，性能远高于CAS**。**

时间戳

时间戳就是在数据库表中单独加一列时间戳，比如“TimeStamp”，每次读出来的时候，把该字 段也读出来，当写回去的时候，把该字段加1，提交之前 ，跟数据库的该字段比较一次，如果比数 据库的值大的话，就允许保存，否则不允许保存，这种处理方法虽然不使用数据库系统提供的锁 机制，但是这种方法可以大大提高数据库处理的并发量，

以上悲观锁所说的加“锁”，其实分为几种锁，分别是：排它锁（写锁）和共享锁（读锁）。

## 3.12. 数据库锁

1. 行级锁

行级锁是一种排他锁，防止其他事务修改此行；在使用以下语句时，Oracle 会自动应用行级锁：

INSERT、UPDATE、DELETE、SELECT … FOR UPDATE [OF columns] [WAIT n | NOWAIT];

SELECT … FOR UPDATE 语句允许用户一次锁定多条记录进行更新

使用 COMMIT 或 ROLLBACK 语句释放锁。

2. 表级锁

表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分 MySQL 引擎支持。最常使 用的 MYISAM 与 INNODB 都支持表级锁定。表级锁定分为表共享读锁（共享锁）与表独占写锁 （排他锁）。

3. 页级锁

页级锁是 MySQL 中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级 冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录。BDB 支持页级锁

## 3.13.基于 Redis 分布式锁

1. 获取锁的时候，使用 setnx（ETNX key val：当且仅当 key 不存在时，set 一个 key 为 val 的字符串，返回 1；若 key 存在，则什么都不做，返回 0）加锁，锁的 value 值为一个随机生成的 UUID，在释放锁的时候进行判断。并使用 expire 命令为锁添 加一个超时时间，超过该时间则自动释放锁。

2. 获取锁的时候调用 setnx，如果返回 0，则该锁正在被别人使用，返回 1 则成功获取 锁。 还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

3. 释放锁的时候，通过 UUID 判断是不是该锁，若是该锁，则执行 delete 进行锁释放。

## 3.14 分区分表

分库分表有垂直切分和水平切分两种。

▪ 垂直切分：将表按照功能模块、关系密切程度划分出来，部署到不同的库上。例如，我们会 建立定义数据库workDB、商品数据库payDB、用户数据库userDB、日志数据库logDB等， 分别用于存储项目数据定义表、商品定义表、用户数据表、日志数据表等。

▪ 水平切分：当一个表中的数据量过大时，我们可以把该表的数据按照某种规则，例如 userID 散列，进行划分，然后存储到多个结构相同的表，和不同的库上。例如，我们的 userDB 中的用户数据表中，每一个表的数据量都很大，就可以把 userDB 切分为结构相同的多个 userDB：part0DB、part1DB 等，再将 userDB 上的用户数据表 userTable，切分为很多 userTable：userTable0、userTable1 等，然后将这些表按照一定的规则存储到多个 userDB 上。

## 3.15 应该使用哪一种方式来实施数据库分库分表，这要看数据库中数据量的瓶颈 所在，并综合项目的业务类型进行考虑。

如果数据库是因为表太多而造成海量数据，并且项目的各项业务逻辑划分清晰、 低耦合，那么规则简单明了、容易实施的垂直切分必是首选。

而如果数据库中的表并不多，但单表的数据量很大、或数据热度很高，这种情况 之下就应该选择水平切分，水平切分比垂直切分要复杂一些，它将原本逻辑上属 于一体的数据进行了物理分割，除了在分割时要对分割的粒度做好评估，考虑数 据平均和负载平均，后期也将对项目人员及应用程序产生额外的数据管理负担。 在现实项目中，往往是这两种情况兼而有之，这就需要做出权衡，甚至既需要垂 直切分，又需要水平切分。我们的游戏项目便综合使用了垂直与水平切分，我们 首先对数据库进行垂直切分，然后，再针对一部分表，通常是用户数据表，进行 水平切分。

单库多表 ：

随着用户数量的增加，user 表的数据量会越来越大，当数据量达到一定 程度的时候对 user 表的查询会渐渐的变慢，从而影响整个 DB 的性能。如果使 用 mysql, 还有一个更严重的问题是，当需要添加一列的时候，mysql 会锁表， 期间所有的读写操作只能等待。

可以将user进行水平的切分，产生两个表结构完全一样的user\_0000,user\_0001 等表，user\_0000 + user\_0001 + …的数据刚好是一份完整的数据。

多库多表 ：

随着数据量增加也许单台 DB 的存储空间不够，随着查询量的增加单台数据 库服务器已经没办法支撑。这个时候可以再对数据库进行水平区分。

分库分表规则举例： 通过分库分表规则查找到对应的表和库的过程。如分库分表的规则是 user\_id 除以 4 的方式，当用户新注册了一个账号，账号 id 的 123,我们可以通过 id 除以 4 的方式确定此账号应该保存到 User\_0003 表中。当用户 123 登录的时 候，我们通过 123 除以 4 后确定记录在 User\_0003 中。

## 3.16 mysql 读写分离

在实际的应用中，绝大部分情况都是读远大于写。Mysql 提供了读写分离 的机制，所有的写操作都必须对应到 Master，读操作可以在 Master 和 Slave 机 器上进行，Slave 与 Master 的结构完全一样，一个 Master 可以有多个 Slave,甚 至 Slave 下还可以挂 Slave,通过此方式可以有效的提高 DB 集群的每秒查询率. 26 所有的写操作都是先在 Master 上操作，然后同步更新到 Slave 上，所以 从 Master 同步到 Slave 机器有一定的延迟，当系统很繁忙的时候，延迟问题会 更加严重，Slave 机器数量的增加也会使这个问题更加严重。

此外，可以看出 Master 是集群的瓶颈，当写操作过多，会严重影响到 Master 的 稳定性，如果 Master 挂掉，整个集群都将不能正常工作。 所以，1. 当读压力很大的时候，可以考虑添加 Slave 机器的分式解决，但是当 Slave 机器达到一定的数量就得考虑分库了。 2. 当写压力很大的时候，就必须 得进行分库操作。

## 3.17 MySQL 常用30种SQL查询语句优化方法

1、应尽量避免在 where 子句中使用!=或<>操作符，否则将引擎放弃使用索引而进行全表扫描。

2、对查询进行优化，应尽量避免全表扫描，首先应考虑在 where 及 order by 涉及的列上建立索引。

3、应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num is null

可以在num上设置默认值0，确保表中num列没有null值，然后这样查询：

select id from t where num=0

4、尽量避免在 where 子句中使用 or 来连接条件，否则将导致引擎放弃使用索引而进行全表扫描，如：

select id from t where num=10 or num=20

可以这样查询：

select id from t where num=10

union all

select id from t where num=20

5、下面的查询也将导致全表扫描：(不能前置百分号)

select id from t where name like ‘%c%’

下面走索引

select id from t where name like ‘c%’

若要提高效率，可以考虑全文检索。

6、in 和 not in 也要慎用，否则会导致全表扫描，如：

select id from t where num in(1,2,3)

对于连续的数值，能用 between 就不要用 in 了：select id from t where num between 1 and 3

7、如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

select id from t where num=@num

可以改为强制查询使用索引：

select id from t with(index(索引名)) where num=@num

8、应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where num/2=100

应改为:

select id from t where num=100\*2

9、应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

select id from t where substring(name,1,3)=’abc’ –name以abc开头的id

select id from t where datediff(day,createdate,’2005-11-30′)=0 –’2005-11-30′生成的id

应改为:

select id from t where name like ‘abc%’

select id from t where createdate>=’2005-11-30′ and createdate<’2005-12-1′

10、不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

11、在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使 用，并且应尽可能的让字段顺序与索引顺序相一致。

12、不要写一些没有意义的查询，如需要生成一个空表结构：

select col1,col2 into #t from t where 1=0

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

create table #t(…)

13、很多时候用 exists 代替 in 是一个好的选择：

select num from a where num in(select num from b)

用下面的语句替换：

select num from a where exists(select 1 from b where num=a.num)

14、并不是所有索引对查询都有效，SQL是根据表中数据来进行查询优化的，当索引列有大量数据重复时，SQL查询可能不会去利用索引，如一表中有字段 sex，male、female几乎各一半，那么即使在sex上建了索引也对查询效率起不了作用。

15、索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数较好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。

16.应尽可能的避免更新 clustered 索引数据列，因为 clustered 索引数据列的顺序就是表记录的物理存储顺序，一旦该列值改变将导致整个表记录的顺序的调整，会耗费相当大的资源。若应用系统需要频繁更新 clustered 索引数据列，那么需要考虑是否应将该索引建为 clustered 索引。

17、尽量使用数字型字段，若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。这是因为引擎在处理查询和连接时会 逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

18、尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

19、任何地方都不要使用 select \* from t ，用具体的字段列表代替“\*”，不要返回用不到的任何字段。

20、尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

21、避免频繁创建和删除临时表，以减少系统表资源的消耗。

22、临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件，较好使 用导出表。

23、在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

24、如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

25、尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

26、使用基于游标的方法或临时表方法之前，应先寻找基于集的解决方案来解决问题，基于集的方法通常更有效。

27、与临时表一样，游标并不是不可使用。对小型数据集使用 FAST\_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

28、在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONEINPROC 消息。

29、尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

30、尽量避免大事务操作，提高系统并发能力。

# SpringMVC框架

1、什么是Spring MVC ？简单介绍下你对springMVC的理解?

Spring MVC是一个基于Java的实现了MVC设计模式的请求驱动类型的轻量级Web框架，通过把Model，View，Controller分离，将web层进行职责解耦，把复杂的web应用分成逻辑清晰的几部分，简化开发，减少出错，方便组内开发人员之间的配合。

2、SpringMVC的流程？



（1）用户发送请求至前端控制器DispatcherServlet；

（2） DispatcherServlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handle；

（3）处理器映射器根据请求url找到具体的处理器，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet；

（4）DispatcherServlet 调用 HandlerAdapter处理器适配器；

（5）HandlerAdapter 经过适配调用 具体处理器(Handler，也叫后端控制器)；

（6）Handler执行完成返回ModelAndView；

（7）HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet；

（8）DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析；

（9）ViewResolver解析后返回具体View；

（10）DispatcherServlet对View进行渲染视图（即将模型数据填充至视图中）

（11）DispatcherServlet响应用户。

3、Springmvc的优点:

（1）可以支持各种视图技术,而不仅仅局限于JSP；

（2）与Spring框架集成（如IoC容器、AOP等）；

（3）清晰的角色分配：前端控制器(dispatcherServlet) , 请求到处理器映射（handlerMapping), 处理器适配器（HandlerAdapter), 视图解析器（ViewResolver）。

（4） 支持各种请求资源的映射策略。

4、Spring MVC的主要组件？

（1）前端控制器 DispatcherServlet（不需要程序员开发）

作用：接收请求、响应结果，相当于转发器，有了DispatcherServlet 就减少了其它组件之间的耦合度。

（2）处理器映射器HandlerMapping（不需要程序员开发）

作用：根据请求的URL来查找Handler

（3）处理器适配器HandlerAdapter

注意：在编写Handler的时候要按照HandlerAdapter要求的规则去编写，这样适配器HandlerAdapter才可以正确的去执行Handler。

（4）处理器Handler（需要程序员开发）

（5）视图解析器 ViewResolver（不需要程序员开发）

作用：进行视图的解析，根据视图逻辑名解析成真正的视图（view）

（6）视图View（需要程序员开发jsp）

View是一个接口， 它的实现类支持不同的视图类型（jsp，freemarker，pdf等等）

5、springMVC和struts2的区别有哪些?

（1）springmvc的入口是一个servlet即前端控制器（DispatchServlet），而struts2入口是一个filter过虑器（StrutsPrepareAndExecuteFilter）。

（2）springmvc是基于方法开发(一个url对应一个方法)，请求参数传递到方法的形参，可以设计为单例或多例(建议单例)，struts2是基于类开发，传递参数是通过类的属性，只能设计为多例。

（3）Struts采用值栈存储请求和响应的数据，通过OGNL存取数据，springmvc通过参数解析器是将request请求内容解析，并给方法形参赋值，将数据和视图封装成ModelAndView对象，最后又将ModelAndView中的模型数据通过reques域传输到页面。Jsp视图解析器默认使用jstl。

6、SpringMVC怎么样设定重定向和转发的？

（1）转发：在返回值前面加"forward:"，譬如"forward:user.do?name=method4"

（2）重定向：在返回值前面加"redirect:"，譬如"redirect:http://www.baidu.com"

7、SpringMvc怎么和AJAX相互调用的？

通过Jackson框架就可以把Java里面的对象直接转化成Js可以识别的Json对象。具体步骤如下 ：

（1）加入Jackson.jar

（2）在配置文件中配置json的映射

（3）在接受Ajax方法里面可以直接返回Object,List等,但方法前面要加上@ResponseBody注解。

8、如何解决POST请求中文乱码问题，GET的又如何处理呢？

（1）解决post请求乱码问题：

在web.xml中配置一个CharacterEncodingFilter过滤器，设置成utf-8；

<filter>

<filter-name>CharacterEncodingFilter</filter-name>

  <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>

    <init-param>

 <param-name>encoding</param-name>

<param-value>utf-8</param-value>

</init-param>

</filter>

<filter-mapping>

    <filter-name>CharacterEncodingFilter</filter-name>

    <url-pattern>/\*</url-pattern>

</filter-mapping>

（2）get请求中文参数出现乱码解决方法有两个：

①修改tomcat配置文件添加编码与工程编码一致，如下：

<ConnectorURIEncoding="utf-8" connectionTimeout="20000" port="8080" protocol="HTTP/1.1" redirectPort="8443"/>

 ②另外一种方法对参数进行重新编码：

String userName = new String(request.getParamter("userName").getBytes("ISO8859-1"),"utf-8")

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码。

9、Spring MVC的异常处理 ？

答：可以将异常抛给Spring框架，由Spring框架来处理；我们只需要配置简单的异常处理器，在异常处理器中添视图页面即可。

10、SpringMvc的控制器是不是单例模式,如果是,有什么问题,怎么解决？

答：是单例模式,所以在多线程访问的时候有线程安全问题,不要用同步,会影响性能的,解决方案是在控制器里面不能写字段。

11、 SpringMVC常用的注解有哪些？

@RequestMapping：用于处理请求 url 映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

@RequestBody：注解实现接收http请求的json数据，将json转换为java对象。

@ResponseBody：注解实现将conreoller方法返回对象转化为json对象响应给客户。

12、SpingMvc中的控制器的注解一般用那个,有没有别的注解可以替代？

答：一般用@Conntroller注解,表示是表现层,不能用别的注解代替。

13、如果在拦截请求中，我想拦截get方式提交的方法,怎么配置？

答：可以在@RequestMapping注解里面加上method=RequestMethod.GET。

14、怎样在方法里面得到Request,或者Session？

答：直接在方法的形参中声明request,SpringMvc就自动把request对象传入。

15、如果想在拦截的方法里面得到从前台传入的参数,怎么得到？

答：直接在形参里面声明这个参数就可以,但必须名字和传过来的参数一样。

16、如果前台有很多个参数传入,并且这些参数都是一个对象的,那么怎么样快速得到这个对象？

答：直接在方法中声明这个对象,SpringMvc就自动会把属性赋值到这个对象里面。

17、SpringMvc中函数的返回值是什么？

答：返回值可以有很多类型,有String, ModelAndView。ModelAndView类把视图和数据都合并的一起的，但一般用String比较好。

18、SpringMvc用什么对象从后台向前台传递数据的？

答：通过ModelMap对象,可以在这个对象里面调用put方法,把对象加到里面,前台就可以通过el表达式拿到。

19、怎么样把ModelMap里面的数据放入Session里面？

答：可以在类上面加上@SessionAttributes注解,里面包含的字符串就是要放入session里面的key。

20、SpringMvc里面拦截器是怎么写的：

有两种写法,一种是实现HandlerInterceptor接口，另外一种是继承适配器类，接着在接口方法当中，实现处理逻辑；然后在SpringMvc的配置文件中配置拦截器即可：

<!-- 配置SpringMvc的拦截器 -->

<mvc:interceptors>

<!-- 配置一个拦截器的Bean就可以了 默认是对所有请求都拦截 -->

<bean id="myInterceptor" class="com.zwp.action.MyHandlerInterceptor"></bean>

<!-- 只针对部分请求拦截 -->

<mvc:interceptor>

<mvc:mapping path="/modelMap.do" />

<bean class="com.zwp.action.MyHandlerInterceptorAdapter" />

</mvc:interceptor>

</mvc:interceptors>

21、注解原理：

注解本质是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。我们通过反射获取注解时，返回的是Java运行时生成的动态代理对象。通过代理对象调用自定义注解的方法，会最终调用AnnotationInvocationHandler的invoke方法。该方法会从memberValues这个Map中索引出对应的值。而memberValues的来源是Java常量池。

# Spring框架

## 1. Spring是什么?

        Spring是一个轻量级的IoC和AOP容器框架。是为Java应用程序提供基础性服务的一套框架，目的是用于简化企业应用程序的开发，它使得开发者只需要关心业务需求。常见的配置方式有三种：基于XML的配置、基于注解的配置、基于Java的配置。

主要由以下几个模块组成：

Spring Core：核心类库，提供IOC服务；

Spring Context：提供框架式的Bean访问方式，以及企业级功能（JNDI、定时任务等）；

Spring AOP：AOP服务；

Spring DAO：对JDBC的抽象，简化了数据访问异常的处理；

Spring ORM：对现有的ORM框架的支持；

Spring Web：提供了基本的面向Web的综合特性，例如多方文件上传；

Spring MVC：提供面向Web应用的Model-View-Controller实现。

## 2.Spring 的优点？

（1）spring属于低侵入式设计，代码的污染极低；

（2）spring的DI机制将对象之间的依赖关系交由框架处理，减低组件的耦合性；

（3）Spring提供了AOP技术，支持将一些通用任务，如安全、事务、日志、权限等进行集中式管理，从而提供更好的复用。

（4）spring对于主流的应用框架提供了集成支持。

## 3 Spring的AOP理解？

OOP面向对象，允许开发者定义纵向的关系，但并适用于定义横向的关系，导致了大量代码的重复，而不利于各个模块的重用。

AOP，一般称为面向切面，作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。

AOP实现的关键在于 代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

（1）AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。

（2）Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。

Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

        ①JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler 通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例,  生成目标类的代理对象。

        ②如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

（3）静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

 InvocationHandler 的 invoke(Object  proxy,Method  method,Object[] args)：proxy是最终生成的代理实例;  method 是被代理目标实例的某个具体方法;  args 是被代理目标实例某个方法的具体入参, 在方法反射调用时使用。

## 4 Spring的IoC理解？

（1）IOC就是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到Spring容器中，并由容器根据配置文件去创建实例和管理各个实例之间的依赖关系，对象与对象之间松散耦合，也利于功能的复用。DI依赖注入，和控制反转是同一个概念的不同角度的描述，即 应用程序在运行时依赖IoC容器来动态注入对象需要的外部资源。

（2）最直观的表达就是，IOC让对象的创建不用去new了，可以由spring自动生产，使用java的反射机制，根据配置文件在运行时动态的去创建对象以及管理对象，并调用对象的方法的。

（3）Spring的IOC有三种注入方式 ：构造器注入、setter方法注入、根据注解注入。

IoC让相互协作的组件保持松散的耦合，而AOP编程允许你把遍布于应用各层的功能分离出来形成可重用的功能组件。

## 5 BeanFactory和ApplicationContext有什么区别？

        BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

（1）BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

①继承MessageSource，因此支持国际化。

②统一的资源文件访问方式。

③提供在监听器中注册bean的事件。

④同时加载多个配置文件。

⑤载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

（2）①BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

        ②ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。 ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean ,确保当你需要的时候，你就不用等待，因为它们已经创建好了。

        ③相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

（3）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader。

（4）BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

## 6 请解释Spring Bean的生命周期？

 首先说一下Servlet的生命周期：实例化，初始init，接收请求service，销毁destroy；

 Spring上下文中的Bean生命周期也类似，如下：

（1）实例化Bean：

对于BeanFactory容器，当客户向容器请求一个尚未初始化的bean时，或初始化bean的时候需要注入另一个尚未初始化的依赖时，容器就会调用createBean进行实例化。对于ApplicationContext容器，当容器启动结束后，通过获取BeanDefinition对象中的信息，实例化所有的bean。

（2）设置对象属性（依赖注入）：

实例化后的对象被封装在BeanWrapper对象中，紧接着，Spring根据BeanDefinition中的信息 以及 通过BeanWrapper提供的设置属性的接口完成依赖注入。

（3）处理Aware接口：

接着，Spring会检测该对象是否实现了xxxAware接口，并将相关的xxxAware实例注入给Bean：

①如果这个Bean已经实现了BeanNameAware接口，会调用它实现的setBeanName(String beanId)方法，此处传递的就是Spring配置文件中Bean的id值；

②如果这个Bean已经实现了BeanFactoryAware接口，会调用它实现的setBeanFactory()方法，传递的是Spring工厂自身。

③如果这个Bean已经实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文；

（4）BeanPostProcessor：

如果想对Bean进行一些自定义的处理，那么可以让Bean实现了BeanPostProcessor接口，那将会调用postProcessBeforeInitialization(Object obj, String s)方法。由于这个方法是在Bean初始化结束时调用的，所以可以被应用于内存或缓存技术；

（5）InitializingBean 与 init-method：

如果Bean在Spring配置文件中配置了 init-method 属性，则会自动调用其配置的初始化方法。

（6）如果这个Bean实现了BeanPostProcessor接口，将会调用postProcessAfterInitialization(Object obj, String s)方法；

以上几个步骤完成后，Bean就已经被正确创建了，之后就可以使用这个Bean了。

（7）DisposableBean：

当Bean不再需要时，会经过清理阶段，如果Bean实现了DisposableBean这个接口，会调用其实现的destroy()方法；

（8）destroy-method：

最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

## 7 解释Spring支持的几种bean的作用域。

Spring容器中的bean可以分为5个范围：

（1）singleton：默认，每个容器中只有一个bean的实例，单例的模式由BeanFactory自身来维护。

（2）prototype：为每一个bean请求提供一个实例。

（3）request：为每一个网络请求创建一个实例，在请求完成以后，bean会失效并被垃圾回收器回收。

（4）session：与request范围类似，确保每个session中有一个bean的实例，在session过期后，bean会随之失效。

（5）global-session：全局作用域，global-session和Portlet应用相关。当你的应用部署在Portlet容器中工作时，它包含很多portlet。如果你想要声明让所有的portlet共用全局的存储变量的话，那么这全局变量需要存储在global-session中。全局作用域与Servlet中的session作用域效果相同。

## 8 使用注解之前要开启自动扫描功能

其中base-package为需要扫描的包(含子包)。

|  |  |
| --- | --- |
| 1 | <context:component-scan base-package="cn.test"/> |

@Configuration把一个类作为一个IoC容器，它的某个方法头上如果注册了@Bean，就会作为这个Spring容器中的Bean。  
@Scope注解 作用域  
@Lazy(true) 表示延迟初始化  
@Service用于标注业务层组件、   
@Controller用于标注控制层组件（如struts中的action）  
@Repository用于标注数据访问组件，即DAO组件。  
@Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注。  
@Scope用于指定scope作用域的（用在类上）

---------------jsr250----  
@PostConstruct用于指定初始化方法（用在方法上）  
@PreDestory用于指定销毁方法（用在方法上）

@Resource 默认按名称装配，当找不到与名称匹配的bean才会按类型装配。

----------  
**@DependsOn：定义Bean初始化及销毁时的顺序  
@Primary：自动装配时当出现多个Bean候选者时，被注解为@Primary的Bean将作为首选者，否则将抛出异常**@Autowired 默认按类型装配，如果我们想使用按名称装配，可以结合@Qualifier注解一起使用。如下：  
@Autowired @Qualifier("personDaoBean") 存在多个实例配合使用

## 9 Spring框架中的单例Beans是线程安全的么？

        Spring框架并没有对单例bean进行任何多线程的封装处理。关于单例bean的线程安全和并发问题需要开发者自行去搞定。但实际上，大部分的Spring bean并没有可变的状态(比如Serview类和DAO类)，所以在某种程度上说Spring的单例bean是线程安全的。如果你的bean有多种状态的话（比如 View Model 对象），就需要自行保证线程安全。最浅显的解决办法就是将多态bean的作用域由“singleton”变更为“prototype”。

## 10 Spring如何处理线程并发问题？

在一般情况下，只有无状态的Bean才可以在多线程环境下共享，在Spring中，绝大部分Bean都可以声明为singleton作用域，因为Spring对一些Bean中非线程安全状态采用ThreadLocal进行处理，解决线程安全问题。

ThreadLocal和线程同步机制都是为了解决多线程中相同变量的访问冲突问题。同步机制采用了“时间换空间”的方式，仅提供一份变量，不同的线程在访问前需要获取锁，没获得锁的线程则需要排队。而ThreadLocal采用了“空间换时间”的方式。

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。ThreadLocal提供了线程安全的共享对象，在编写多线程代码时，可以把不安全的变量封装进ThreadLocal。

## 11.1、Spring基于xml注入bean的几种方式：

（1）Set方法注入；

（2）构造器注入：①通过index设置参数的位置；②通过type设置参数类型；

（3）静态工厂注入；

（4）实例工厂；

详细内容可以阅读：<https://blog.csdn.net/a745233700/article/details/89307518>

## 11.2、Spring的自动装配：

在spring中，对象无需自己查找或创建与其关联的其他对象，由容器负责把需要相互协作的对象引用赋予各个对象，使用autowire来配置自动装载模式。

在Spring框架xml配置中共有5种自动装配：

（1）no：默认的方式是不进行自动装配的，通过手工设置ref属性来进行装配bean。

（2）byName：通过bean的名称进行自动装配，如果一个bean的 property 与另一bean 的name 相同，就进行自动装配。

（3）byType：通过参数的数据类型进行自动装配。

（4）constructor：利用构造函数进行装配，并且构造函数的参数通过byType进行装配。

（5）autodetect：自动探测，如果有构造方法，通过 construct的方式自动装配，否则使用 byType的方式自动装配。

基于注解的方式：

使用@Autowired注解来自动装配指定的bean。在使用@Autowired注解之前需要在Spring配置文件进行配置，<context:annotation-config />。在启动spring IoC时，容器自动装载了一个AutowiredAnnotationBeanPostProcessor后置处理器，当容器扫描到@Autowied、@Resource或@Inject时，就会在IoC容器自动查找需要的bean，并装配给该对象的属性。在使用@Autowired时，首先在容器中查询对应类型的bean：

如果查询结果刚好为一个，就将该bean装配给@Autowired指定的数据；

如果查询的结果不止一个，那么@Autowired会根据名称来查找；

如果上述查找的结果为空，那么会抛出异常。解决方法时，使用required=false。

@Autowired可用于：构造函数、成员变量、Setter方法

注：@Autowired和@Resource之间的区别

(1) @Autowired默认是按照类型装配注入的，默认情况下它要求依赖对象必须存在（可以设置它required属性为false）。

(2) @Resource默认是按照名称来装配注入的，只有当找不到与名称匹配的bean才会按照类型来装配注入。

## 12 Spring 框架中都用到了哪些设计模式？

（1）工厂模式：BeanFactory就是简单工厂模式的体现，用来创建对象的实例；

（2）单例模式：Bean默认为单例模式。

（3）代理模式：Spring的AOP功能用到了JDK的动态代理和CGLIB字节码生成技术；

（4）模板方法：用来解决代码重复的问题。比如. RestTemplate, JmsTemplate, JpaTemplate。

（5）观察者模式：定义对象键一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都会得到通知被制动更新，如Spring中listener的实现--ApplicationListener。

## 13 Spring事务的实现方式和实现原理：

Spring事务的本质其实就是数据库对事务的支持，没有数据库的事务支持，spring是无法提供事务功能的。真正的数据库层的事务提交和回滚是通过binlog或者redo log实现的。

**（1）Spring事务的种类：**

spring支持编程式事务管理和声明式事务管理两种方式：

①编程式事务管理使用TransactionTemplate。

②声明式事务管理建立在AOP之上的。其本质是通过AOP功能，对方法前后进行拦截，将事务处理的功能编织到拦截的方法中，也就是在目标方法开始之前加入一个事务，在执行完目标方法之后根据执行情况提交或者回滚事务。

声明式事务最大的优点就是不需要在业务逻辑代码中掺杂事务管理的代码，只需在配置文件中做相关的事务规则声明或通过@Transactional注解的方式，便可以将事务规则应用到业务逻辑中。

声明式事务管理要优于编程式事务管理，这正是spring倡导的非侵入式的开发方式，使业务代码不受污染，只要加上注解就可以获得完全的事务支持。唯一不足地方是，最细粒度只能作用到方法级别，无法做到像编程式事务那样可以作用到代码块级别。

**（2）spring的事务传播行为：**

spring事务的传播行为说的是，当多个事务同时存在的时候，spring如何处理这些事务的行为。

① PROPAGATION\_REQUIRED：如果当前没有事务，就创建一个新事务，如果当前存在事务，就加入该事务，该设置是最常用的设置。

② PROPAGATION\_SUPPORTS：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就以非事务执行。‘

③ PROPAGATION\_MANDATORY：支持当前事务，如果当前存在事务，就加入该事务，如果当前不存在事务，就抛出异常。

④ PROPAGATION\_REQUIRES\_NEW：创建新事务，无论当前存不存在事务，都创建新事务。

⑤ PROPAGATION\_NOT\_SUPPORTED：以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

⑥ PROPAGATION\_NEVER：以非事务方式执行，如果当前存在事务，则抛出异常。

⑦ PROPAGATION\_NESTED：如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则按REQUIRED属性执行。

**（3）Spring中的隔离级别：**

① ISOLATION\_DEFAULT：这是个 PlatfromTransactionManager 默认的隔离级别，使用数据库默认的事务隔离级别。

② ISOLATION\_READ\_UNCOMMITTED：读未提交，允许另外一个事务可以看到这个事务未提交的数据。

③ ISOLATION\_READ\_COMMITTED：读已提交，保证一个事务修改的数据提交后才能被另一事务读取，而且能看到该事务对已有记录的更新。

④ ISOLATION\_REPEATABLE\_READ：可重复读，保证一个事务修改的数据提交后才能被另一事务读取，但是不能看到该事务对已有记录的更新。

⑤ ISOLATION\_SERIALIZABLE：一个事务在执行的过程中完全看不到其他事务对数据库所做的更新。

## 14 Spring框架中有哪些不同类型的事件？

Spring 提供了以下5种标准的事件：

（1）上下文更新事件（ContextRefreshedEvent）：在调用ConfigurableApplicationContext 接口中的refresh()方法时被触发。

（2）上下文开始事件（ContextStartedEvent）：当容器调用ConfigurableApplicationContext的Start()方法开始/重新开始容器时触发该事件。

（3）上下文停止事件（ContextStoppedEvent）：当容器调用ConfigurableApplicationContext的Stop()方法停止容器时触发该事件。

（4）上下文关闭事件（ContextClosedEvent）：当ApplicationContext被关闭时触发该事件。容器被关闭时，其管理的所有单例Bean都被销毁。

（5）请求处理事件（RequestHandledEvent）：在Web应用中，当一个http请求（request）结束触发该事件。

如果一个bean实现了ApplicationListener接口，当一个ApplicationEvent 被发布以后，bean会自动被通知。

## 15 解释一下Spring AOP里面的几个名词：

（1）切面（Aspect）：被抽取的公共模块，可能会横切多个对象。 在Spring AOP中，切面可以使用通用类（基于模式的风格） 或者在普通类中以 @AspectJ 注解来实现。

（2）连接点（Join point）：指方法，在Spring AOP中，一个连接点 总是 代表一个方法的执行。

（3）通知（Advice）：在切面的某个特定的连接点（Join point）上执行的动作。通知有各种类型，其中包括“around”、“before”和“after”等通知。许多AOP框架，包括Spring，都是以拦截器做通知模型， 并维护一个以连接点为中心的拦截器链。

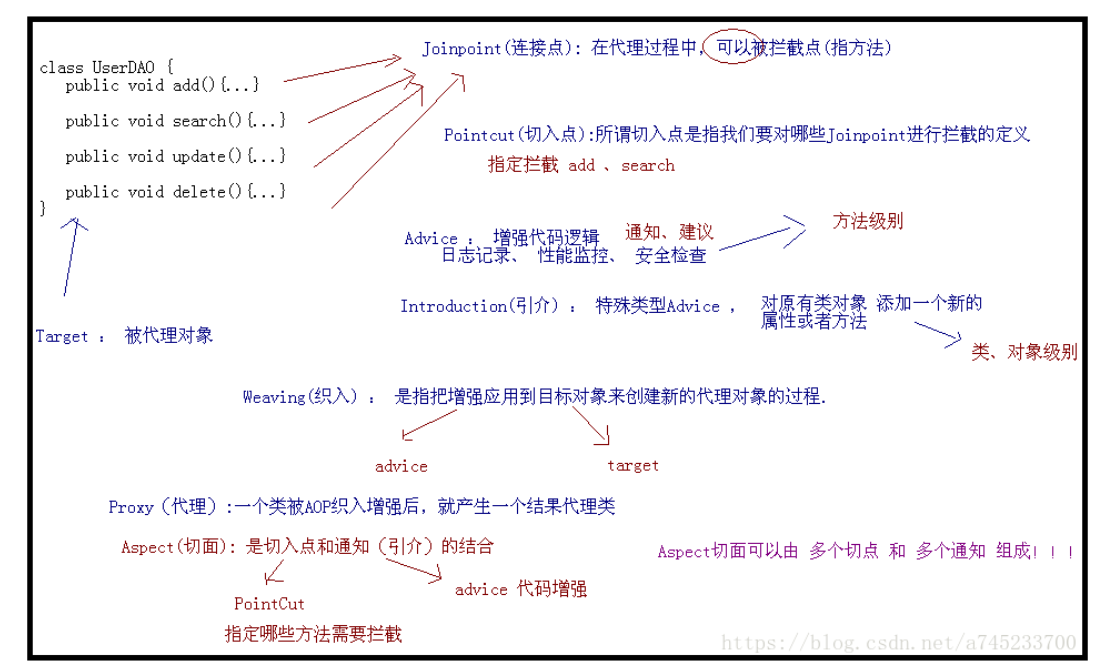
（4）切入点（Pointcut）：切入点是指 我们要对哪些Join point进行拦截的定义。通过切入点表达式，指定拦截的方法，比如指定拦截add\*、search\*。

（5）引入（Introduction）：（也被称为内部类型声明（inter-type declaration））。声明额外的方法或者某个类型的字段。Spring允许引入新的接口（以及一个对应的实现）到任何被代理的对象。例如，你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。

（6）目标对象（Target Object）： 被一个或者多个切面（aspect）所通知（advise）的对象。也有人把它叫做 被通知（adviced） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied） 对象。

（7）织入（Weaving）：指把增强应用到目标对象来创建新的代理对象的过程。Spring是在运行时完成织入。

切入点（pointcut）和连接点（join point）匹配的概念是AOP的关键，这使得AOP不同于其它仅仅提供拦截功能的旧技术。 切入点使得定位通知（advice）可独立于OO层次。 例如，一个提供声明式事务管理的around通知可以被应用到一组横跨多个对象中的方法上（例如服务层的所有业务操作）。



## 16 Spring通知有哪些类型？

（1）前置通知（Before advice）：在某连接点（join point）之前执行的通知，但这个通知不能阻止连接点前的执行（除非它抛出一个异常）。

（2）返回后通知（After returning advice）：在某连接点（join point）正常完成后执行的通知：例如，一个方法没有抛出任何异常，正常返回。

（3）抛出异常后通知（After throwing advice）：在方法抛出异常退出时执行的通知。

（4）后通知（After (finally) advice）：当某连接点退出的时候执行的通知（不论是正常返回还是异常退出）。

（5）环绕通知（Around Advice）：包围一个连接点（join point）的通知，如方法调用。这是最强大的一种通知类型。 环绕通知可以在方法调用前后完成自定义的行为。它也会选择是否继续执行连接点或直接返回它们自己的返回值或抛出异常来结束执行。 环绕通知是最常用的一种通知类型。大部分基于拦截的AOP框架，例如Nanning和JBoss4，都只提供环绕通知。

同一个aspect，不同advice的执行顺序：

①没有异常情况下的执行顺序：

around before advice  
before advice  
target method 执行  
around after advice  
after advice  
afterReturning

②有异常情况下的执行顺序：

around before advice  
before advice  
target method 执行  
around after advice  
after advice  
afterThrowing:异常发生  
java.lang.RuntimeException: 异常发生

# 六 Mybatis框架

**1、什么是Mybatis？**

（1）Mybatis是一个半ORM（对象关系映射）框架，它内部封装了JDBC，开发时只需要关注SQL语句本身，不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。程序员直接编写原生态sql，可以严格控制sql执行性能，灵活度高。

（2）MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。

（3）通过xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过java对象和 statement中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。（从执行sql到返回result的过程）。

**2、Mybaits的优点：**

（1）基于SQL语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL写在XML里，解除sql与程序代码的耦合，便于统一管理；提供XML标签，支持编写动态SQL语句，并可重用。

（2）与JDBC相比，减少了50%以上的代码量，消除了JDBC大量冗余的代码，不需要手动开关连接；

（3）很好的与各种数据库兼容（因为MyBatis使用JDBC来连接数据库，所以只要JDBC支持的数据库MyBatis都支持）。

（4）能够与Spring很好的集成；

（5）提供映射标签，支持对象与数据库的ORM字段关系映射；提供对象关系映射标签，支持对象关系组件维护。

**3、MyBatis框架的缺点：**

（1）SQL语句的编写工作量较大，尤其当字段多、关联表多时，对开发人员编写SQL语句的功底有一定要求。

（2）SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

**4、MyBatis框架适用场合：**

（1）MyBatis专注于SQL本身，是一个足够灵活的DAO层解决方案。

（2）对性能的要求很高，或者需求变化较多的项目，如互联网项目，MyBatis将是不错的选择。

**5、MyBatis与Hibernate有哪些不同？**

（1）Mybatis和hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句。

（2）Mybatis直接编写原生态sql，可以严格控制sql执行性能，灵活度高，非常适合对关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一但需求变化要求迅速输出成果。但是灵活的前提是mybatis无法做到数据库无关性，如果需要实现支持多种数据库的软件，则需要自定义多套sql映射文件，工作量大。

（3）Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用hibernate开发可以节省很多代码，提高效率。

**6、#{}和${}的区别是什么？**

#{}是预编译处理，${}是字符串替换。

Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；

Mybatis在处理${}时，就是把${}替换成变量的值。

使用#{}可以有效的防止SQL注入，提高系统安全性。

## ****7、Mybatis是如何进行分页的？分页插件的原理是什么？****

Mybatis使用RowBounds对象进行分页，它是针对ResultSet结果集执行的内存分页，而非物理分页。可以在sql内直接书写带有物理分页的参数来完成物理分页功能，也可以使用分页插件来完成物理分页。

分页插件的基本原理是使用Mybatis提供的插件接口，实现自定义插件，在插件的拦截方法内拦截待执行的sql，然后重写sql，根据dialect方言，添加对应的物理分页语句和物理分页参数。

## ****8、Mybatis是如何将sql执行结果封装为目标对象并返回的？都有哪些映射形式？****

第一种是使用<resultMap>标签，逐一定义数据库列名和对象属性名之间的映射关系。

第二种是使用sql列的别名功能，将列的别名书写为对象属性名。

有了列名与属性名的映射关系后，Mybatis通过反射创建对象，同时使用反射给对象的属性逐一赋值并返回，那些找不到映射关系的属性，是无法完成赋值的。

## ****9、Mybatis动态sql有什么用？执行原理？有哪些动态sql？****

Mybatis动态sql可以在Xml映射文件内，以标签的形式编写动态sql，执行原理是根据表达式的值 完成逻辑判断并动态拼接sql的功能。

Mybatis提供了9种动态sql标签：trim | where | set | foreach | if | choose | when | otherwise | bind。

## ****10、Xml映射文件中，除了常见的select|insert|updae|delete标签之外，还有哪些标签？****

答：<resultMap>、<parameterMap>、<sql>、<include>、<selectKey>，加上动态sql的9个标签，其中<sql>为sql片段标签，通过<include>标签引入sql片段，<selectKey>为不支持自增的主键生成策略标签。

## ****11、Mybatis的Xml映射文件中，不同的Xml映射文件，id是否可以重复？****

不同的Xml映射文件，如果配置了namespace，那么id可以重复；如果没有配置namespace，那么id不能重复；

原因就是namespace+id是作为Map<String, MapperStatement>的key使用的，如果没有namespace，就剩下id，那么，id重复会导致数据互相覆盖。有了namespace，自然id就可以重复，namespace不同，namespace+id自然也就不同。

## ****12、为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？****

Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

## ****13、MyBatis实现一对一有几种方式?具体怎么操作的？****

有联合查询和嵌套查询,联合查询是几个表联合查询,只查询一次, 通过在resultMap里面配置association节点配置一对一的类就可以完成；

嵌套查询是先查一个表，根据这个表里面的结果的 外键id，去再另外一个表里面查询数据,也是通过association配置，但另外一个表的查询通过select属性配置。

## ****14、MyBatis实现一对多有几种方式,怎么操作的？****

有联合查询和嵌套查询。联合查询是几个表联合查询,只查询一次,通过在resultMap里面的collection节点配置一对多的类就可以完成；嵌套查询是先查一个表,根据这个表里面的 结果的外键id,去再另外一个表里面查询数据,也是通过配置collection,但另外一个表的查询通过select节点配置。

## ****15、Mybatis是否支持延迟加载？如果支持，它的实现原理是什么？****

答：Mybatis仅支持association关联对象和collection关联集合对象的延迟加载，association指的就是一对一，collection指的就是一对多查询。在Mybatis配置文件中，可以配置是否启用延迟加载lazyLoadingEnabled=true|false。

它的原理是，使用CGLIB创建目标对象的代理对象，当调用目标方法时，进入拦截器方法，比如调用a.getB().getName()，拦截器invoke()方法发现a.getB()是null值，那么就会单独发送事先保存好的查询关联B对象的sql，把B查询上来，然后调用a.setB(b)，于是a的对象b属性就有值了，接着完成a.getB().getName()方法的调用。这就是延迟加载的基本原理。

当然了，不光是Mybatis，几乎所有的包括Hibernate，支持延迟加载的原理都是一样的。

## ****16、Mybatis的一级、二级缓存:****

1）一级缓存: 基于 PerpetualCache 的 HashMap 本地缓存，其存储作用域为 Session，当 Session flush 或 close 之后，该 Session 中的所有 Cache 就将清空，默认打开一级缓存。

2）二级缓存与一级缓存其机制相同，默认也是采用 PerpetualCache，HashMap 存储，不同在于其存储作用域为 Mapper(Namespace)，并且可自定义存储源，如 Ehcache。默认不打开二级缓存，要开启二级缓存，使用二级缓存属性类需要实现Serializable序列化接口(可用来保存对象的状态),可在它的映射文件中配置<cache/> ；

3）对于缓存数据更新机制，当某一个作用域(一级缓存 Session/二级缓存Namespaces)的进行了C/U/D 操作后，默认该作用域下所有 select 中的缓存将被 clear。

## ****17、什么是MyBatis的接口绑定？有哪些实现方式？****

接口绑定，就是在MyBatis中任意定义接口,然后把接口里面的方法和SQL语句绑定, 我们直接调用接口方法就可以,这样比起原来了SqlSession提供的方法我们可以有更加灵活的选择和设置。

接口绑定有两种实现方式,一种是通过注解绑定，就是在接口的方法上面加上 @Select、@Update等注解，里面包含Sql语句来绑定；另外一种就是通过xml里面写SQL来绑定, 在这种情况下,要指定xml映射文件里面的namespace必须为接口的全路径名。当Sql语句比较简单时候,用注解绑定, 当SQL语句比较复杂时候,用xml绑定,一般用xml绑定的比较多。

## ****18、使用MyBatis的mapper接口调用时有哪些要求？****

①  Mapper接口方法名和mapper.xml中定义的每个sql的id相同；  
②  Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql 的parameterType的类型相同；  
③  Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同；  
④  Mapper.xml文件中的namespace即是mapper接口的类路径。

## ****19、简述Mybatis的插件运行原理，以及如何编写一个插件。****

答：Mybatis仅可以编写针对ParameterHandler、ResultSetHandler、StatementHandler、Executor这4种接口的插件，Mybatis使用JDK的动态代理，为需要拦截的接口生成代理对象以实现接口方法拦截功能，每当执行这4种接口对象的方法时，就会进入拦截方法，具体就是InvocationHandler的invoke()方法，当然，只会拦截那些你指定需要拦截的方法。

编写插件：实现Mybatis的Interceptor接口并复写intercept()方法，然后在给插件编写注解，指定要拦截哪一个接口的哪些方法即可，记住，别忘了在配置文件中配置你编写的插件。

# SpringBoot Springcloud框架

### 1. 传统开发一般是基于什么框架进行开发的？

传统开发开发方式我们一般采用SSH或者SSM进行开发。随着互联网的发展和社会业务场景的复杂性增加。为了能够更加高效快捷的开发，我们慢慢趋于使用springboot进行开发。但针对一些传统行业(比如：银行项目，政府项目，或者比较保密的项目我们还是使用传统，因为他们使用时间更久更稳定。)

SSH 通常指的是 Struts2 做控制器(controller)，spring 管理各层的组件，hibernate 负责持久化层。

SSM 则指的是 SpringMVC 做控制器(controller)，Spring 管理各层的组件，MyBatis 负责持久化层。

共同点：1.Spring依赖注入DI来管理各层的组件。2.使用面向切面编程AOP管理事物、日志、权限等。

不同点：1.Struts2 和 SpringMVC 控制器(controller)控制视图和模型的交互机制的不同，

Struts2是Action类级别，SpringMVC是方法级别，更容易实现RESTful风格。

### 2. 什么是SOA的？

SOA，对于刚接触企业的工作的朋友来说，可能不大了解这个概念。下面就通过一些讲解来跟大家简单分享一下SOA。SOA，面向服务的体系结构（Service-Oriented Architecture）。它的主要作用是将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。  
  接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。

**简而言之：SOA是一种微服务的架构，是一种架构思想，把我们项目中涉及业务模块做更小的拆分，降低耦合，一遍满足客户更多，更广的需求。基于他的思想我们一般会引入dubbo消息中间件，或者springcloud框架。**

### 什么是微服务？

#### 3.1微服务的概念

微服务架构风格是一种将单个应用程序作为一套小型服务开发的方法，每种应用程序都在自己的进程中运行，并与轻量级机制（通常是HTTP资源API）进行通信。 这些服务是围绕业务功能构建的，可以通过全自动部署机制独立部署。 这些服务的集中管理最少，可以用不同的编程语言编写，并使用不同的数据存储技术。

**简而言之：微服务是把我们要做的项目根据业务进行独立部署，可以把同一项目中不同语言编写的业务基于Restful风格开发接口进行网络间通讯。**

#### 3.2 微服务的特点

**1.独立部署，灵活扩展**

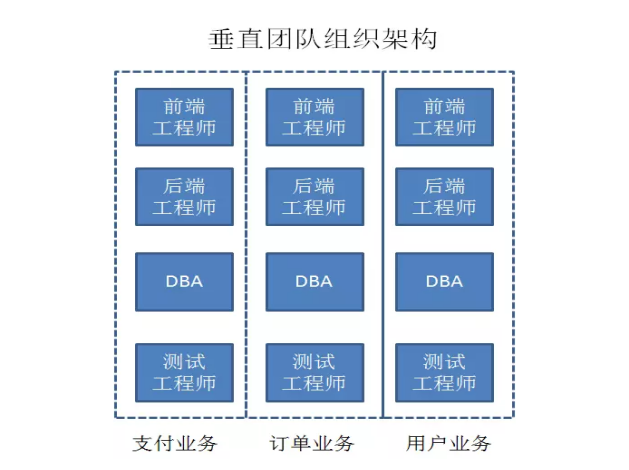
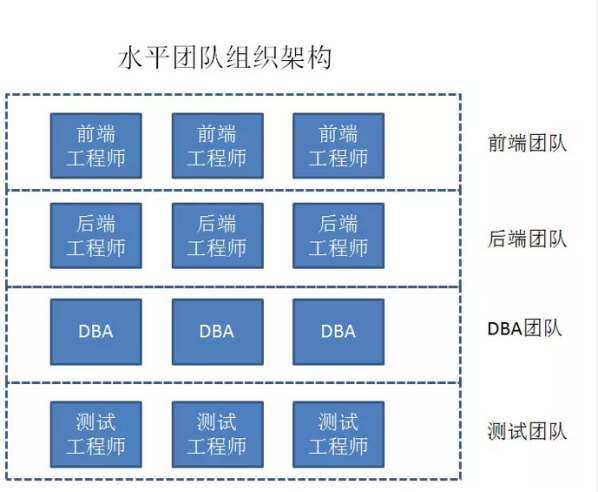
传统的单体架构是以整个系统为单位进行部署，而微服务则是以每一个独立组件（例如用户服务，商品服务）为单位进行部署。什么意思呢？比如根据每个服务的吞吐量不同，支付服务需要部署20台机器，用户服务需要部署30台机器，而商品服务只需要部署10台机器。这种灵活部署只有微服务架构才能实现。而近几年流行的Docker，为微服务架构提供了有效的容器。

**2.资源的有效隔离**

微服务设计的原则之一，就是每一个微服务拥有独立的数据源，假如微服务A想要读写微服务B的数据库，只能调用微服务B对外暴露的接口来完成。这样有效避免了服务之间争用数据库和缓存资源所带来的问题。

**3.团队组织架构的调整**

微服务设计的思想也改变了原有的企业研发团队组织架构。传统的研发组织架构是水平架构，前端有前端的团队，后端有后端的团队，DBA有DBA的团队，测试有测试的团队。而**微服务的设计思想对团队的划分有着一定的影响，使得团队组织架构的划分更倾向于垂直架构，比如用户业务是一个团队来负责，支付业务是一个团队来负责。**



#### 3.3 为什么要使用微服务架构，微服务的优缺点是什么？

单一架构模式在项目初期的时候开发,测试,部署方便,但是随着项目逐步开发,项目工程会很大,最终互相之间会有繁琐的jar包。

　　　　1.不再适用于敏捷开发,任何开发人员都不能完全理解,在修复漏洞和添加新功能时候变得复杂

　　　　2.项目模块越大,启动越慢,很小的改动也需要重新部署整个项目

　　　　3.任何模块的漏洞,都会降低系统的可靠性

　　　　4.如果想整体使用新的技术,新的框架,那是不可能的

　　如果采用微服务,解决了单一系统的复杂性,主要有以下优点

　　　　1.可采用敏捷开发模式,选择合适的技术

　　　　2.每一个微服务是独立部署的,可以快速迭代部署

　　　　3.一些需要负载均衡的服务可以使用ngix同一反向代理分发请求,这样不需要整个系统负载均衡了

微服务的缺点:

　　　　1.微服务作为分布式系统时候带来了复杂性,当应用是整体应用时,模块之间可以内部调用,但微服务是多个独一的应用,调用起来增加了一定的复杂性

　　　　2.多个数据库,事物的实现会头痛

　　　　3.测试微服务变得复杂,当一个服务依赖于另一个服务时候,测试时需要另一个服务的支持

　　　　4.整体系统只需要部署在一组相同的服务器上,而微服务需要更高的自动化形似

### 3.1 SOA和微服务的区别？

**SOA（面向服务的架构）**：面向服务的架构（SOA）是一个组件模型，它将应用程序的不同功能单元（称为服务）通过这些服务之间定义良好的接口和契约联系起来。接口是采用中立的方式进行定义的，它应该独立于实现服务的硬件平台、操作系统和编程语言。这使得构建在各种各样的系统中的服务可以以一种统一和通用的方式进行交互。

**微服务：**微服务架构是一种将单个应用程序作为一套小型服务开发的方法，每种应用程序都在自己的进程中运行，并与轻量级机制（通常是HTTP资源API）进行通信。这些服务是围绕业务功能构建的，可以通过全自动部署机制独立部署。这些服务的集中管理最少，可以用不同的编程语言编写，并使用不同的数据存储技术。

SOA和微服务的主要区别：

         微服务剔除SOA中复杂的ESB企业服务总线，所有的业务智能逻辑在服务内部处理，使用Http（Rest API）进行轻量化通讯

         SOA强调按水平架构划分为：前、后端、数据库、测试等，微服务强调按垂直架构划分，按业务能力划分，每个服务完成一种特定的功能，服务即产品

         SOA将组件以library的方式和应用部署在同一个进程中运行，微服务则是各个服务独立运行。

         传统应用倾向于使用统一的技术平台来解决所有问题，微服务可以针对不同业务特征选择不同技术平台，去中心统一化，发挥各种技术平台的特长。

         SOA架构强调的是异构系统之间的通信和解耦合；（一种粗粒度、松耦合的服务架构）

         微服务架构强调的是系统按业务边界做细粒度的拆分和部署。

### 4. Springboot是什么？为什么我们选择使用springboot开发？

分析：由于以上的分析，xml 的可读性差，基于注解的方式也有不足，那这时候，为了解决这些问题，更加的提升开发效率 SpringBoot登场了。它的到来 就把我们的xml配置基本上都省略了。但是也不是绝对的完全省略，而是让我们操作应用起来跟简单了。但是它也有

一个弊端，那就是一旦出了问题不好定位。

-- 什么是SpringBoot？

简单的说，springboot就是整合了很多优秀的框架，不用我们自己手动的去写一堆xml配置然后进行配置。

　　从本质上来说，SpringBoot就是Spring,它做了那些没有它你也会去做的Spring Bean配置。它使用“习惯优于配置”（项目中存在大量的配置，此外还内置了一个习惯性的配置，让你无需手动进行配置）的理念让你的项目快速运行起来。使 用Spring Boot很容易创建一个独立运行（运行jar,内嵌Servlet容器）、准生产级别的基于Spring框架的项目，使用Spring Boot你可以不用或者只需要很少的Spring配置。

-- 为什么要用SpringBoot？（敏捷开发）

快速的完成Spring和其他框架的构建整合。进一步提升开发效率。这就是当今流行的敏捷开发。

#### 4.1 springboot的发家史之前情回顾

**1. 回顾 java中 “配置” 的发展历史（引出springboot）**

\*\* 什么是配置文件

\*.ini、\*.properties、\*.xml都是配置文件，这些配置文件往往被奉若神明，自从我们接触电脑的那一天开始，就被人警告不要乱弄他们。其实他们与java的HashMap结构是一样的，都是一个存放key-value对的容器，只是配置文件那些是属性与属性值的key-value对罢了。但是为何不能乱碰它呢？因为这些key-valueyi对一般是记录着程序运行的参数，而且很多是初始化的参数，一般这些参数都是自程序的开始运行，也就是程序的出生到程序被关闭，也就是程序死亡都需要使用的定值。正如我们每一个人的名字、性别、出生地这些属性一样，从我们出生到死亡都要不停地用。

\*\* java中最原始的时候没有任何配置，以及配置的发展历程

概述：在java开发的婴儿期，不需要考虑配置，因为程序很简单，在编码过程中就可以完成几乎所有的需求。

**-- 第一阶段： \*.properties \*.xml配置文件**

随着业务的增强，和代码复杂度的提高，这时候一些公用的部分，以及项目初始化到项目卸载的的运行过程的一些必备参数都抽取到配置文件当中。以上在spring1.x 的时代被广泛应用。

**-- 第二阶段： 注解配置**

伴随着更加复杂的业务，以及代码代码的相互依赖的特性，这时候，庞大的配置文件成了我们头疼的事情。就在这时候，迎着spring2.x时代的到来，也随着jdk发行1.5版本对注解的支持，这时候我们就用注解代替之前的xml配置。

**-- 第三阶段： java配置（基于类的配置）**

又随着时间的推移，在spring3.x的到来，为我们提供了更简单的配置，那就是基于 java的配置。这样一来 我们连spring核心配置文件都省了，只需要在你的类上加一个 @Configuration 注解，就代表当前类是一个可配置的类，接着就可以用@Bean 来配置当前方法返回的是一个bean 实例对象。

总结：综上所述 在几种配置方式中，可以对比一下它们各自的优势和短板。

**-- xml 容易操控**，对于一些经常修改的配置更适合，但是可读性差开发过程中一定程度上限制了开发效率。

**-- 注解** 是一种分散式的元数据，与源代码紧绑定。配置简单，易操作，但是由于注解是分散式的，所以维护起来不易。

**-- java配置** java配置其实就是注解的有一种拓展，目的是为了更加的简单易用，更加的减少xml的配置。

#### 4.2 springboot环境搭建

***步骤分析：***

**建项目--**

创建一个maven工程

注意：springboot整合的时候在建立maven工程的时候一定要创建一个父工程 而且，父工程选择2.x的版本，因为我们用的是spring5.x的版本。

**导依赖--**

spring-boot-web.jar

**写配置**… 配置SpringBoot的核心配置文件application.properties

此处省略一堆字。因为我们今天要用springboot

**创建类--**

在src目录下的com.itiheimaba包下新建一个SpringBootApplicationRunner的类。并且在类上加上 @SpringBootApplication 的注解来表示此类就是一个SpringBoot的启动器。另外还需要我们在此类中写一个main方法，方法中利用

SpringApplication.run（SpringBootApplicationRunner.class,args）;

来完成最终的启动。

***访问localhost：8080***

***分析spring-boot-web这个依赖：由于这个依赖的导入，根据maven的依赖机制他给我们导入了一大堆支持我们开发的依赖。***

#### 4.3 SpringMVC的传统环境搭建

注：在进行SpringBoot和SpringMVC整合之前咱们先回顾一下传统的SpringMVC环境搭建

步骤分析：

1. 建项目

-- 新建maven项目 springmvcfirst

-- 注意 这时候要选择 war包

2. 导依赖

-- servlet-api

-- jsp-api

-- spring-context

-- spring-webmvc

3. 写配置

\*\* 配置web.xml 文件

--配置前端核心控制器DiapatcherServlet

<serverlet>

<serverlet-name>DispatcherServerlet</serverlet-name>

<serverlet-calss>类路径</serverlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:springmvc.xml</param-value>

</init-param>

<load-on-stratup>1</load-onstratup>

</serverlet>

<serverlet-mapping>

<serverlet-name>DispatcherServerlet</serverlet-name>

<url-pattern>/</ url-pattern >

</serverlet-mapping>

\*\*配置springmvc.xml

-- 写入约束文件

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:mvc="http://www.springframework.org/schema/mvc"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="

http://www.springframework.org/schema/beans

http://www.springframework.org/schema/beans/spring-beans.xsd

http://www.springframework.org/schema/mvc

http://www.springframework.org/schema/mvc/spring-mvc.xsd

http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context.xsd">

-- 配置要扫描的包

<context-conponent-scan base-pagkage=”…”></context-conponent-scan>

-- 配置视图解析器

<bean id=”viewResolver” class=”InternalResourceViewResolver”>

-- 前缀

<property name=”prefix” value=”/WEB-INF/pages/”></property>

-- 后缀

<property name=”suffix” value=”.jsp”></property>

</bean>

-- 开启spring对注解mvc的支持

<mvc:annotation-driven/>

4. 创建类

\*\* 在src目录下创建 com.itheima.web包，然后创建TestController

\*\* 在webapp目录下新建 index.jsp

\*\* 在 WEB-INF下创建 pages文件夹，然后创建一个success.jsp

5. 将项目add到 Tomcat 下运行 然后访问测试！！！

#### 4.4 SpringBoot整合SpringMVC的环境搭建

***入门案例：***springboot 整合Spring 和 springMVC

步骤分析：

\*\* 在com.itheima 包下新建一个web.controller包

\*\* 在web.controller下新建 HelloController

\*\* 加入 @RestController 注解就ok 啦！

\*\* 然后通过访问 localhost:8080/…

疑问？

此时考虑 我们原来的 index.jsp 呢？还有请求是在哪发送的？

解释：

当SpringBoot 工程启动的时候，就把我们请求对象和相应对象都封装到Jackson的jar包当中了。它的请求和响应都是利用json传输的。

#### 4.5 SpringBoot原理分析及启动时的一些细节

**1. 原理分析：**

解释此类的作用：

这个类就是我们SpringBoot 整合其他框架的一个启动类，一会所有准备都完事后，只要一运行此类，那么整合就完事了。

解释@SpringBootApplication的作用：

@SpringBootApplication = @Configuration + @EnableAutoConfiguration + @ComponentScan

@Configuration -- 提到@Configuration就要提到他的搭档@Bean。使用这两个注解就可以创建一个简单的spring配置类，可以用来替代相应的xml配置文件。

@EnableAutoConfiguration -- 能够自动配置spring的上下文，试图猜测和配置你想要的bean类，通常会自动根据你的类路径和你的bean定义自动配置。

@ComponentScan -- 会自动扫描指定包下的全部标有@Component的类，并注册成bean

经常见到的几个jdk的元注解：

@Target(ElementType.TYPE) // 当前注解的使用范围

@Retention(RetentionPolicy.RUNTIME) //生命周期

@Documented // 声明在生成doc文档时是否带着注解

@Inherited // 声明是否子类会显示父类的注解

总结：一个@SpringBootApplication注解就搞定了所有事，因为它觉得以上三个注解都是我们固定的配置，所以就给我们省略了。又封装出一个注解来代替，这就是SpringBoot 的核心设计思想。

思考--我们为什么要把SpringBoot工程的启动类创建在base包下？

SpringBoot工程运行时，扫描包来创建容器会扫描启动类所在的包以及它下面的子包。

**1. 细节一分析**

SpringApplication.run（SpringBootApplicationRunner.class,args）;

备注：

-- run方法：

此方法是用于把启动类的字节码传入给SpringBoot的核心容器，通过注解配置解析来创建IOC容器。

-- 参数一：就是启动类的字节码。

-- 参数二：通常情况下就是main方法的args

\*\*拓展：args由内容吗？

验证：进行打印，会出现一个对象的地址

解释：数组默认情况下会打印它的第一个对象的地址。

2. 细节二分析

备注：我们关注控制台，有一个SpringBoot的图标。那这个图标我们可以不让其打印吗？

解释：可以。按照以下操作

SpringBootApplication sba = new SpringBootApplication(class);

sba.setBannerMode(Mode.off);

sba.run(args);

#### 4.6 SpringBoot热部署的使用

1. 重启传统的springmvc项目，访问一把！OK

修改：此时随便在controller类中改改 再访问！不OK

原因：我们改变源码了，但是没有重新部署！！！

解决：重新部署一遍，又OK了！！！

2. 分析：

问题：

总是重新部署势必会花费大量的时间！这时候大胆畅想一下能不能修改源码后不重新部署也能生效呢？

注意：修改源码必须重新编译，重启是重新编译的手段。这个思想是没有问题的，但是我们此时要想的是能不能不重新部署也能完成重新编译！！！

解决：利用SpringBoot的热部署就可以实现。

3. 如何实现SpringBoot的热部署

-- 导入依赖

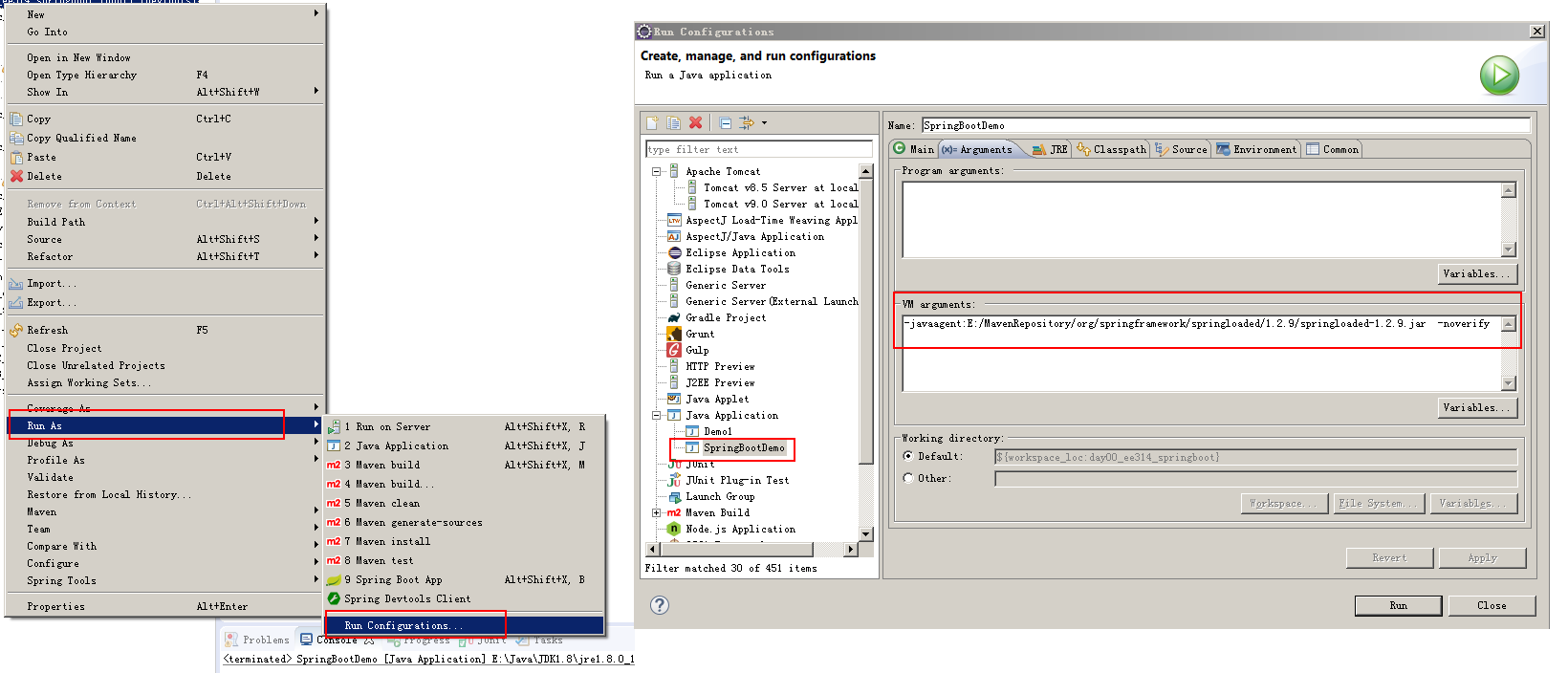
spring-boot-devtools.jar 2.0.0

导完坐标保存后我们的项目在开发工具中会有一个变化



springloaded-1.2.9.jar

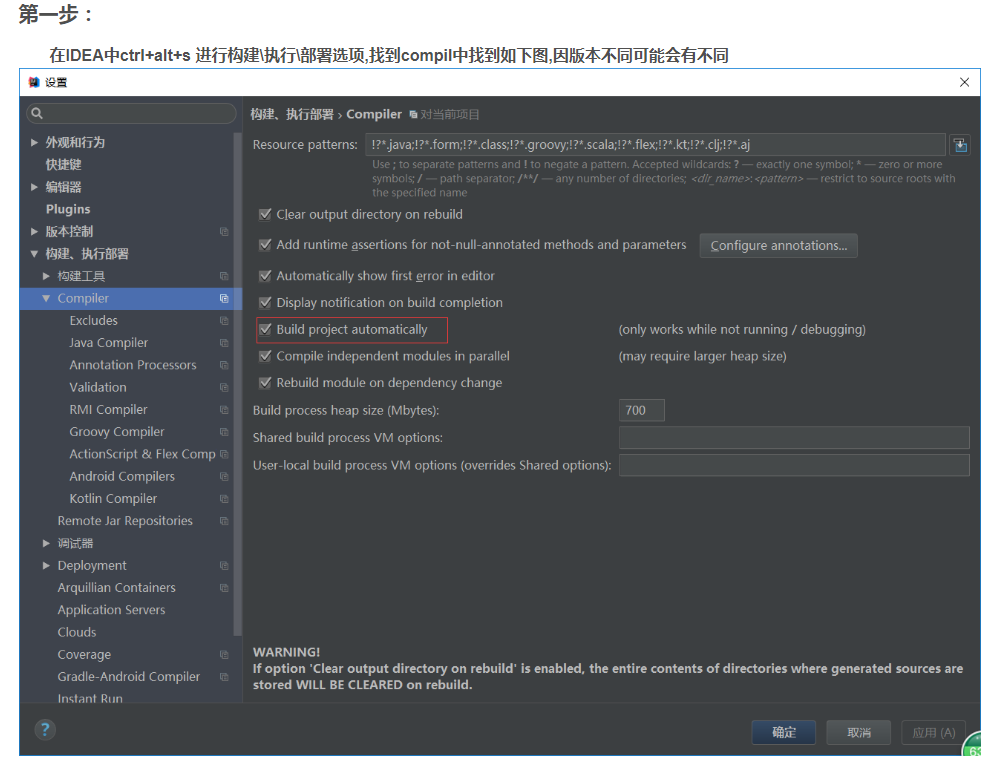
注意：这个jar包我们直接导入不了，需要手动改变一下版本号。导入包以后还需要我们 在eclipse中配置一下如下图：

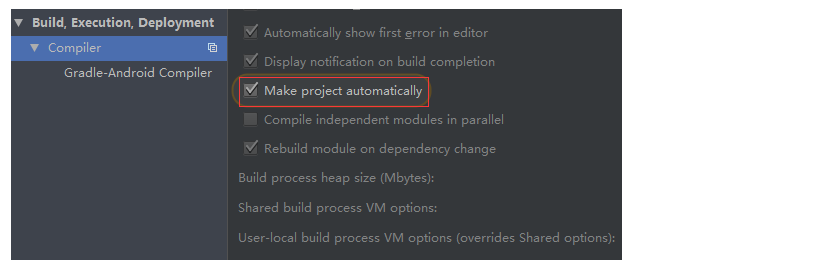


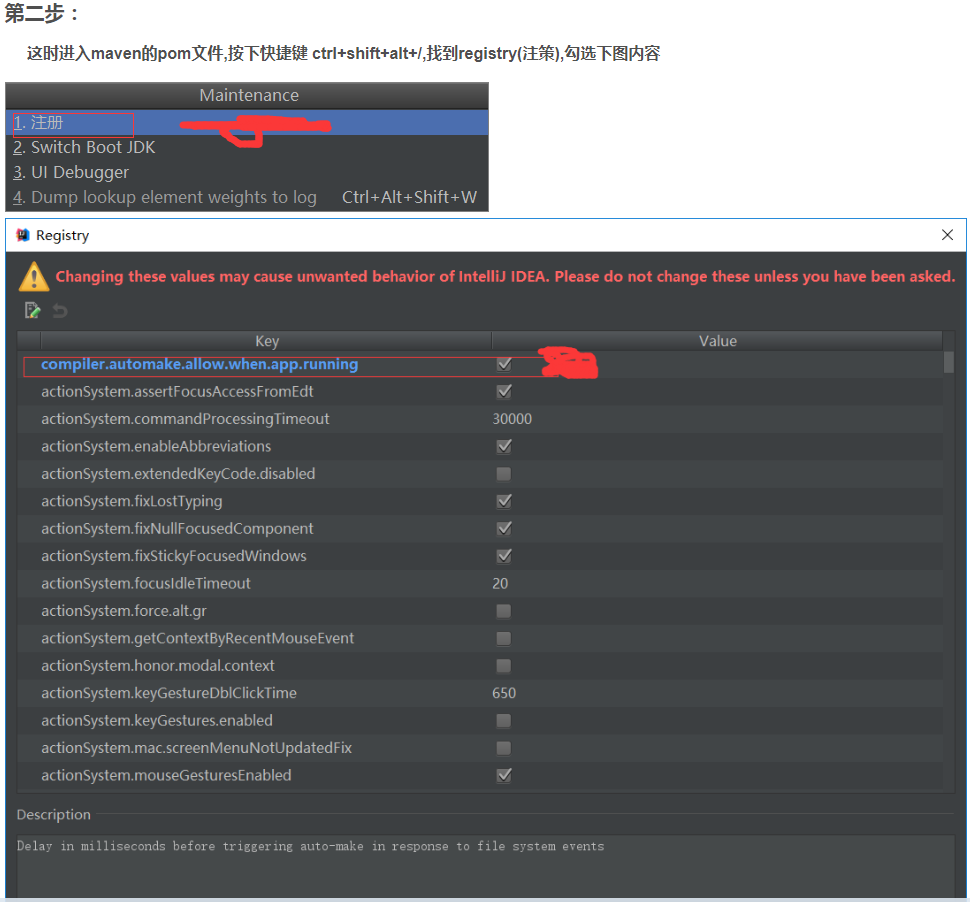
参数配置：是springloaded-1.2.9.jar 这个包在maven本地仓库的位置。

此时热部署就可以正常使用了…

注意：在IDEA中热部署需要设置一下，步骤如下：







最后重启IDEA，导入热部署的 依赖 jar包即可！

#### 4.7 SpringBoot热部署涉之springloaded1.2.9jar包的安装

1. 特别强调！针对springloaded.jar,由于maven中央仓库目前

最高版是1.2.8 没有1.2.9 那我们如何获取这个包呢？

-- 第一步：把springloaded-1.2.9.放到你电脑的任意一个没有中文和空格的目录下路径下。

-- 第二部：然后通过cmd 命令窗口进入 刚才新建的文件夹下。

-- 第三部： 执行maven安装命令

mvn install:install-file

-DgroupId=org.springframework -DartifactId=springloaded

-Dversion=1.2.9

-Dpackaging=jar

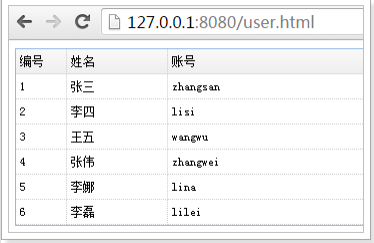
-Dfile=springloaded-1.2.9.BUILD.jar

#### 4.8 SpringBoot整合springdatajpa

1. 需求分析：

使用 Spring Boot + Spring MVC + Spring Data JPA + EasyUI

框架组合实现部门列表查询，效果如下：



2. 步骤分析：

-- 在数据库中创建user表并添加相关数据

在 MySQL 数据库执行以下语句:

DROP TABLE IF EXISTS `user`;

CREATE TABLE `user` (

`id` bigint(20) NOT NULL AUTO\_INCREMENT,

`user\_name` varchar(100) DEFAULT NULL COMMENT '用户名',

`password` varchar(100) DEFAULT NULL COMMENT '密码',

`name` varchar(100) DEFAULT NULL COMMENT '姓名',

PRIMARY KEY (`id`)

) ENGINE=InnoDB AUTO\_INCREMENT=7 DEFAULT CHARSET=utf8;

INSERT INTO `user` VALUES ('1', 'zhangsan', '123456', '张三');

INSERT INTO `user` VALUES ('2', 'lisi', '123456', '李四');

INSERT INTO `user` VALUES ('3', 'wangwu', '123456', '王五');

INSERT INTO `user` VALUES ('4', 'zhangwei', '123456', '张伟');

INSERT INTO `user` VALUES ('5', 'lina', '123456', '李娜');

INSERT INTO `user` VALUES ('6', 'lilei', '123456', '李磊');

-- 创建一个新的项目

在刚才整合完springmvc的基础上进行整合

-- 导依赖

mysql 驱动包

spring-boot-starter-data-jpa.jar

-- 写配置

在项目的resources目录下新建application.properties文件

此文件中配置一些 经常修改 的信息，此处我们配置的是

数据库连接信息。

-- 创建类

\*\* 创建实体类 并利用注解建立与数据库的对应关系。

\*\* 创建Dao接口并且 extends JpaRepository和

JpaSpecificationExecutor

\*\* 创建service层

\*\* 创建Controller层，写相应的查询方法

#### 4.9 加入前端jqueryeasyui的使用

1. 引入jQueryeasyUI 插件，将数据显示到表格中

步骤分析：

-- 在resources目录下新建一个**static文件夹**，必须命名为static 这是springboot的规定，然后将easyUI的相关文件放入，最后直接访问 user.html 通过ajax 请求请求后台数据。

#### 4.10 SpringBoot整合MyBatis

步骤分析：

1. 建项目（还用刚才的）  
2. 导依赖

在 pom.xml 中加入以下依赖

<!-- SpringBoot 的 Mybatis 启动器 -->

<dependency>

<groupId>org.mybatis.spring.boot</groupId>

<artifactId>mybatis-spring-boot-starter</artifactId>

<version>1.1.1</version>

</dependency>

3. 写配置（springboot的核心配置文件，此时不用改！）

4. 创建类

-- 新建IUserMapper 接口 写一个根据username模糊查

询User集合的接口

和之前的方式一样，只是多了两个注解：

@Mapper：声明 Mapper 接口

@Select：声明这个接口所需要使用的 sql，当然，有查询的注解，肯定就有增删改的注解。

import java.util.List;

import org.apache.ibatis.annotations.Mapper;

import org.apache.ibatis.annotations.Select;

import cn.itcast.info.pojo.User;

@Mapper

public interface UserMapper {

@Select("select \* from user where name like

'%${value}%'")

public List<User> queryUserByName(String name);

}

-- Service层新增一个接口和实现方法

-- 进入controller 新增一个方法（此时需要传入参数，我

们用一下REST 风格）

@RequestMapping(“findUserByName/{name}”)

public List<User> findUserByName(@PtahVarable(“name”)String name){

List<User> list = userService.findUserByName(name);

Return list;

}

总结：

通过以上操作我们发现一个问题，就是热部署也不是万能的，当我们修改了Dao层中注解里面的SQL语句并不能自动部署。这个问题要注意，它属于一个细节，但是往往细节也能决定你的身价！！

#### 4.11 SpringBoot整合Redis

1. 准备工作

-- Windows下开启Redis服务

-- 为了方便查查看，开启Redis的一个客户端管理工具

2. 整合步骤分析

-- 建项目（此处还用原来的项目）

-- 导依赖

在 pom.xml 加入依赖

<!-- 配置使用 redis 启动器 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-redis</artifactId>

</dependency>

-- 写配置（注意：SpringBoot的核心配置不用改，但是此时由于是要缓存数据库，需要配置若干注解）

-- 打开SpringBoot启动类，加入@EnableCaching 意思就是开启缓存的支持。

-- 接着到service中在配置缓存的具体适用规则。

\*\* @Cacheable(value=”…”)

注：在执行查询的方法上加@Cacheable注解表示第一次查询到数据库取值然后在缓存中存一份，当再次查询的时候就先到缓存中取值，知道执行缓存清理操作缓存数据才消失，其中重点分析注解中的两个参数 value :指定存入redis中的key的值。

\*\*@CacheEvict(value=”…”)

注：这个注解适用于清除缓存，里面的参数和存入注解时的一样，只是多一个allEntries属性表示是否彻底清除。

#### 4.12 SpringBoot整合Junit

1. 步骤分析：

-- 建项目（此处还用以上的项目为例）

-- 导依赖

在 pom.xml 中加入测试依赖

<!-- 配置测试启动器 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

-- 写配置（SpringBoot核心配置不用改）

-- 创建类

在工程的test目录下新建一个测试类

测试类内容如下：

//替换运行器

@RunWith(SpringJUnit4ClassRunner.class)

//指定引导类的字节码

@SpringBootTest(classes=SpringBootApplicationRunner.class)

public class SpringBootJunitTest {

@Autowired

private IUserService userService;

@Test

public void testFindAll() {

List<User> users = userService.findAll();

for(User user : users) {

System.out.println(user);

}

}

}

总结：SpringBoot的Junit整合其实和spring对Junit的整合差不多，只是换了一个注解配置。

#### 4.13 SpringBoot读取application.properties配置文件

1. properties文件通常是做什么的？

在我们程序中，一些我们经常要进行修改的属性，

通常配在properties文件中，这样当修改的时候就可以修改properties文件就可以了。但是有一个环节必不可少，那就是我们想用这些属性的时候，必然要去读取properties文件。所以读取properties文件也是我们程序中必须的一部分。

2. 在SpringBoot中如何读取核心配置文件application.properties

步骤分析：

-- 在测试类中注入一个Envrionment 对象。

private Envrionment environment;

-- 然后调用这个对象的getProperty() 方法就可以了。

environment.getProperty(key);

注意：此对象只用于读取SpringBoot的核心配置文件，

不是它核心配置文件的内容读不出来。

#### 4.14 SpringBoot读取自定义properties文件

1. 需求分析：

-- 自定义一个mail.properties 文件

-- 再创建一个MailProperty.java 对象

-- mail.properties文件中的值和MailProperty.java对象的属性对应。-- 然后最终直接操作MailProperty.java对象就相当于操作mail.properties文件，从而达到不直接读取mail.properties文件就能对其进行操作的目的。

2. 操作步骤：

-- 自定义一个mail.properties 文件

-- 再创建一个MailProperty.java 对象

-- mail.properties文件中的值和MailProperty.java

对象的属性对应。

-- 在MailProperty.java 对象上配置两个注解

@ConfigurationProperties(prefix=”mail”)

@PropertySource(value=”classpath:mail.properties”)

注意：@ConfigurationProperties prefix属性的含义就是指定properties文件中的属性的前缀。

@PropertySource 注解是指定对应的properties文件的位置。

-- 配置完了还不能用，此时我们需要一个springboot的一个依赖 spring-boot-configuration-processor

-- 完成以上步骤后，发现还不行，此时我们还需要一个注解的配合才能完成。

@Component 或者 @Configuration

把以上注解配置到我们的MailProperty.java上就相当于把这个类当做一个可配置的类，才能够最终将 MailProperty.java 注入到IOC 容器中。

总结：这样做的好处就是当项目中有大量的properties文件的时候我们可以不用频繁的解析配置文件，而是转化成操作对象的方式，变得更简单。

#### 4.15 SpringBoot读取yml配置文件

概述：yml文件也是springboot为我们提供的一种核心配置文件的格式，它除了能像application.properties文件那样写配置，另外他也提供了一种新的数据结构就是更加 一目了然 的树形结构。

用法：

1.读取普通数据

-- 使用@Value 注解配合 Spring 的 EL 表达式

@Value("${mail.host}")

private String mailHost;

2.读取对象数据

-- 和读取自定义的properties一样。

### 5. Springcloud是什么？

#### 5.1 概念：

Spring Cloud 是一系列框架的有序集合，它利用 Spring Boot 的开发便利性简化了分布式系统的开发，比如服务发现、服务网关、服务路由、链路追踪等。其设计目的是为了简化 Spring 应用的搭建和开发过程。该框架遵循“约定大于配置”原则，采用特定的方式进行配置，从而使开发者无需定义大量的 XML 配置。通过这种方式，Spring Boot 致力于在蓬勃发展的快速应用开发领域成为领导者。Spring Cloud 并不重复造轮子，而是将市面上开发得比较好的模块集成进去，进行封装，从而减少了各模块的开发成本。换句话说：Spring Cloud 提供了构建分布式系统所需的“全家桶”。核心组件:Eureka、Ribbon、Feign、Hystrix、Zuul.

**Eureka**：各个服务启动时，Eureka Client都会将服务注册到Eureka Server，并且Eureka Client还可以反过来从Eureka Server拉取注册表，从而知道其他服务在哪里

**Ribbon**：服务间发起请求的时候，基于Ribbon做负载均衡，从一个服务的多台机器中选择一台

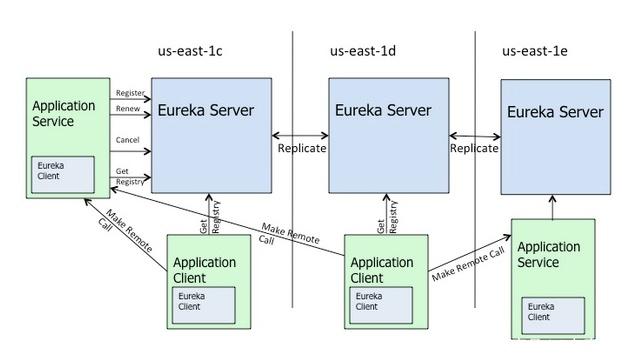
**Feign**：基于Feign的动态代理机制，根据注解和选择的机器，拼接请求URL地址，发起请求

**Hystrix**：发起请求是通过Hystrix的线程池来走的，不同的服务走不同的线程池，实现了不同服务调用的隔离，避免了服务雪崩的问题

**Zuul**：如果前端、移动端要调用后端系统，统一从Zuul网关进入，由Zuul网关转发请求给对应的服务

#### 5.2 核心组件讲解：

##### 5.2.1 Eureka(服务中心)



**作用：实现服务治理（服务注册与发现）**

简介：Spring Cloud Eureka是Spring Cloud Netflix项目下的服务治理模块。

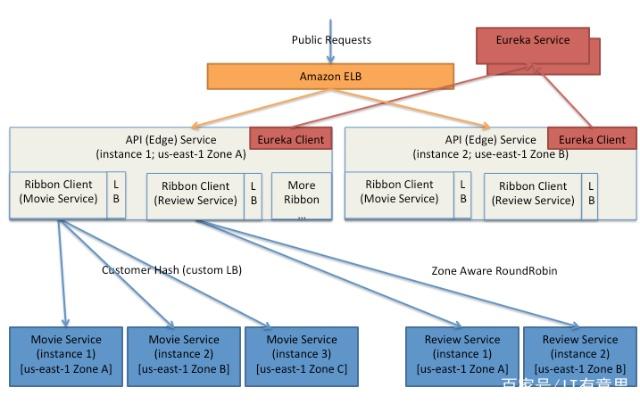
由两个组件组成：Eureka服务端和Eureka客户端。

Eureka服务端用作服务注册中心。支持集群部署。

Eureka客户端是一个java客户端，用来处理服务注册与发现。

在应用启动时，Eureka客户端向服务端注册自己的服务信息，同时将服务端的服务信息缓存到本地。客户端会和服务端周期性的进行心跳交互，以更新服务租约和服务信息。

##### 5.2.2 Ribbon(负载均衡工具)

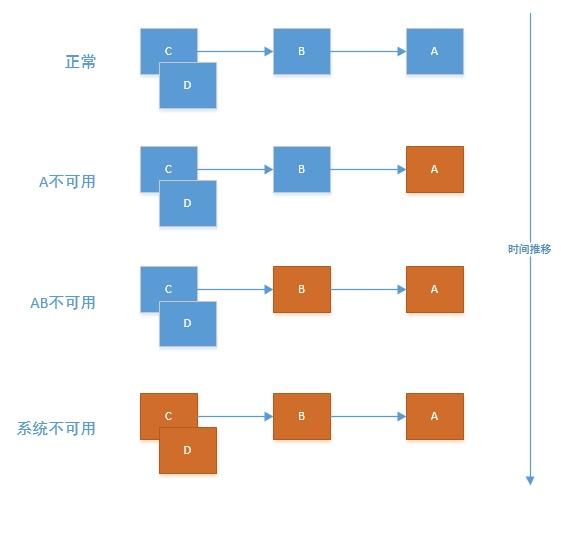


**作用：Ribbon，主要提供客户侧的软件负载均衡算法。**

简介：Spring Cloud Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。通过Spring Cloud的封装，可以让我们轻松地将面向服务的REST模版请求自动转换成客户端负载均衡的服务调用。

注意看上图，关键点就是将外界的rest调用，根据负载均衡策略转换为微服务调用。Ribbon有比较多的负载均衡策略，以后专门讲解。

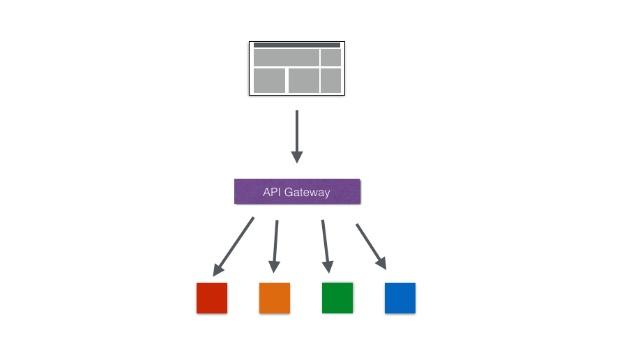
##### 5.2.3 Hystrix(熔断器)



**作用：断路器，保护系统，控制故障范围。**

简介：为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。

##### 5.2.4 Zuul(服务网关)

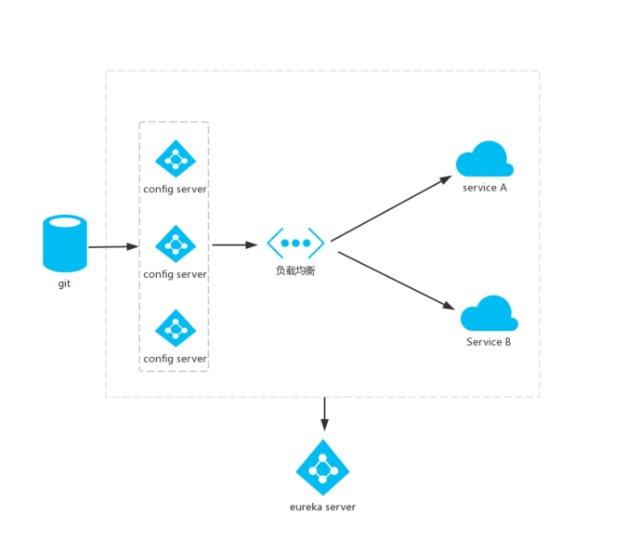


**作用：api网关，路由，负载均衡等多种作用**

简介：类似nginx，反向代理的功能，不过netflix自己增加了一些配合其他组件的特性。

在微服务架构中，后端服务往往不直接开放给调用端，而是通过一个API网关根据请求的url，路由到相应的服务。当添加API网关后，在第三方调用端和服务提供方之间就创建了一面墙，这面墙直接与调用方通信进行权限控制，后将请求均衡分发给后台服务端。

##### 5.2.5 Config(配置中心)



**作用：配置管理**

简介：SpringCloud Config提供服务器端和客户端。服务器存储后端的默认实现使用git，因此它轻松支持标签版本的配置环境，以及可以访问用于管理内容的各种工具。

这个还是静态的，得配合Spring Cloud Bus实现动态的配置更新。

#### 5.3 Springcloud基于idea进行开发的流程

具体搭建步骤建议参考：<https://blog.csdn.net/zhou199252/article/details/80745151>

#### 5.4 Springcloud基于业务场景对于各个组件的讲解

先来给大家说一个业务场景，假设咱们现在开发一个电商网站，要实现支付订单的功能，流程如下：

创建一个订单之后，如果用户立刻支付了这个订单，我们需要将订单状态更新为“已支付”

扣减相应的商品库存 通知仓储中心，进行发货 给用户的这次购物增加相应的积分

针对上述流程，**我们需要有订单服务、库存服务、仓储服务、积分服务**。整个流程的大体思路如下：

用户针对一个订单完成支付之后，就会去找订单服务，更新订单状态

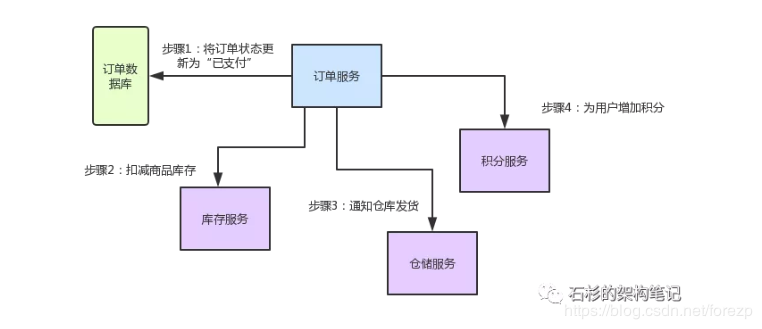
订单服务调用库存服务，完成相应功能

订单服务调用仓储服务，完成相应功能

订单服务调用积分服务，完成相应功能

**至此，整个支付订单的业务流程结束**

下图这张图，清晰表明了各服务间的调用过程：



好！有了业务场景之后，咱们就一起来看看Spring Cloud微服务架构中，这几个组件如何相互协作，各自发挥的作用以及其背后的原理。

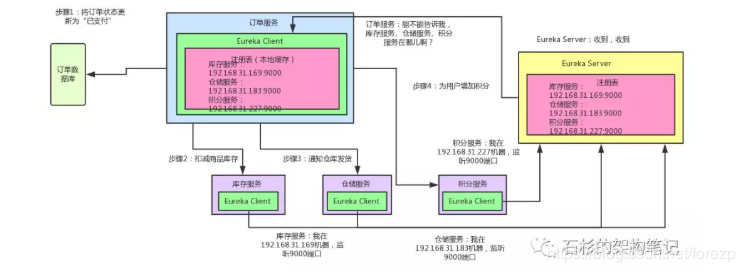
**Spring Cloud核心组件：Eureka**

**咱们来考虑第一个问题**：订单服务想要调用库存服务、仓储服务，或者是积分服务，怎么调用？

订单服务压根儿就不知道人家库存服务在哪台机器上啊！他就算想要发起一个请求，都不知道发送给谁，有心无力！

这时候，就轮到Spring Cloud Eureka出场了。Eureka是微服务架构中的注册中心，专门负责服务的注册与发现。

 咱们来看看下面的这张图，结合图来仔细剖析一下整个流程：



如上图所示，库存服务、仓储服务、积分服务中都有一个Eureka Client组件，这个组件专门负责将这个服务的信息注册到Eureka Server中。说白了，就是告诉Eureka Server，自己在哪台机器上，监听着哪个端口。而Eureka Server是一个注册中心，里面有一个注册表，保存了各服务所在的机器和端口号

订单服务里也有一个Eureka Client组件，这个Eureka Client组件会找Eureka Server问一下：库存服务在哪台机器啊？监听着哪个端口啊？仓储服务呢？积分服务呢？然后就可以把这些相关信息从Eureka Server的注册表中拉取到自己本地缓存起来。

这时如果订单服务想要调用库存服务，不就可以找自己本地的Eureka Client问一下库存服务在哪台机器？监听哪个端口吗？收到响应后，紧接着就可以发送一个请求过去，调用库存服务扣减库存的那个接口！同理，如果订单服务要调用仓储服务、积分服务，也是如法炮制。

总结一下：

**Eureka Client：**负责将这个服务的信息注册到Eureka Server中

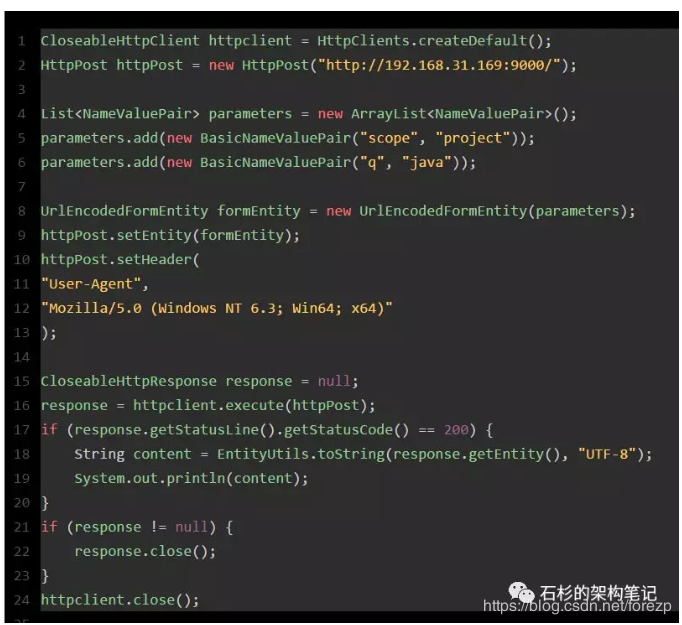
**Eureka Server：**注册中心，里面有一个注册表，保存了各个服务所在的机器和端口号

**Spring Cloud核心组件：Feign**

现在订单服务确实知道库存服务、积分服务、仓库服务在哪里了，同时也监听着哪些端口号了。**但是新问题又来了：难道订单服务要自己写一大堆代码，跟其他服务建立网络连接，然后构造一个复杂的请求，接着发送请求过去，最后对返回的响应结果再写一大堆代码来处理吗？**

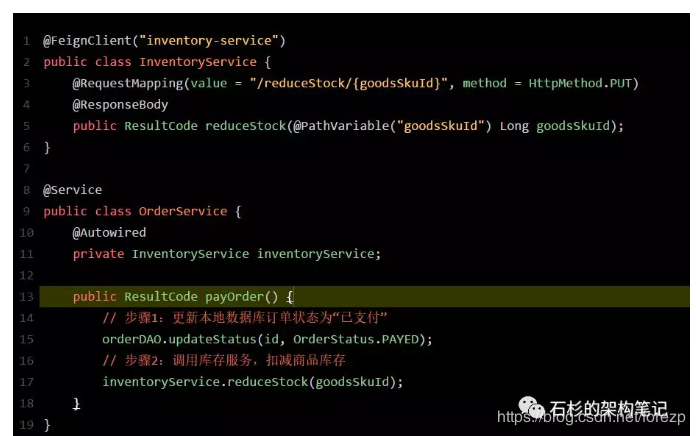
这是上述流程翻译的代码片段，咱们一起来看看，体会一下这种绝望而无助的感受！！！

**友情提示，前方高能：**



看完上面那一大段代码，有没有感到后背发凉、一身冷汗？实际上你进行服务间调用时，如果每次都手写代码，代码量比上面那段要多至少几倍，所以这个事儿压根儿就不是地球人能干的。

既然如此，那怎么办呢？别急，Feign早已为我们提供好了优雅的解决方案。来看看如果用Feign的话，你的订单服务调用库存服务的代码会变成啥样？



看完上面的代码什么感觉？是不是感觉整个世界都干净了，又找到了活下去的勇气！没有底层的建立连接、构造请求、解析响应的代码，直接就是用注解定义一个 FeignClient接口，然后调用那个接口就可以了。人家Feign Client会在底层根据你的注解，跟你指定的服务建立连接、构造请求、发起靕求、获取响应、解析响应，等等。这一系列脏活累活，人家Feign全给你干了。

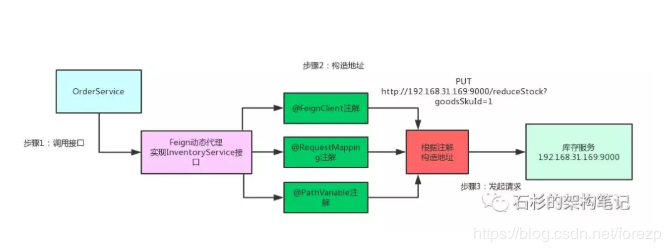
那么问题来了，Feign是如何做到这么神奇的呢？很简单，**Feign的一个关键机制就是使用了动态代理**。咱们一起来看看下面的图，结合图来分析：

首先，如果你对某个接口定义了@FeignClient注解，Feign就会针对这个接口创建一个动态代理

接着你要是调用那个接口，本质就是会调用 Feign创建的动态代理，这是核心中的核心

Feign的动态代理会根据你在接口上的@RequestMapping等注解，来动态构造出你要请求的服务的地址

最后针对这个地址，发起请求、解析响应



Spring Cloud核心组件：Ribbon

说完了Feign，还没完。现在新的问题又来了，如果人家库存服务部署在了5台机器上，如下所示：

192.168.169:9000

192.168.170:9000

192.168.171:9000

192.168.172:9000

192.168.173:9000

**这下麻烦了！人家Feign怎么知道该请求哪台机器呢？**

这时Spring Cloud Ribbon就派上用场了。Ribbon就是专门解决这个问题的。它的作用是负载均衡，会帮你在每次请求时选择一台机器，均匀的把请求分发到各个机器上

Ribbon的负载均衡默认使用的最经典的Round Robin轮询算法。这是啥？简单来说，就是如果订单服务对库存服务发起10次请求，那就先让你请求第1台机器、然后是第2台机器、第3台机器、第4台机器、第5台机器，接着再来—个循环，第1台机器、第2台机器。。。以此类推。

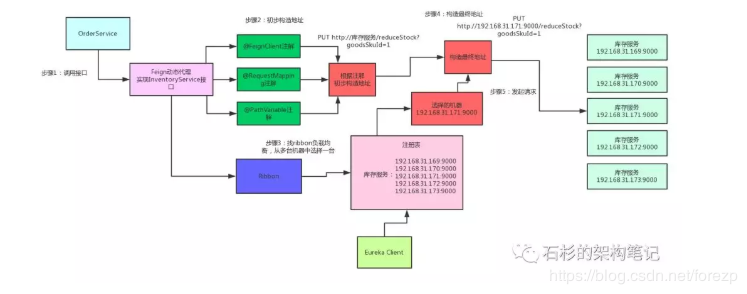
**此外，Ribbon是和Feign以及Eureka紧密协作，完成工作的，具体如下：**

首先Ribbon会从 Eureka Client里获取到对应的服务注册表，也就知道了所有的服务都部署在了哪些机器上，在监听哪些端口号。

然后Ribbon就可以使用默认的Round Robin算法，从中选择一台机器

Feign就会针对这台机器，构造并发起请求。

对上述整个过程，再来一张图，帮助大家更深刻的理解：



**Spring Cloud核心组件：Hystrix**

在微服务架构里，一个系统会有很多的服务。以本文的业务场景为例：订单服务在一个业务流程里需要调用三个服务。现在假设订单服务自己最多只有100个线程可以处理请求，然后呢，积分服务不幸的挂了，每次订单服务调用积分服务的时候，都会卡住几秒钟，然后抛出—个超时异常。

**咱们一起来分析一下，这样会导致什么问题？**

如果系统处于高并发的场景下，大量请求涌过来的时候，订单服务的100个线程都会卡在请求积分服务这块。导致订单服务没有一个线程可以处理请求

然后就会导致别人请求订单服务的时候，发现订单服务也挂了，不响应任何请求了

上面这个，就是**微服务架构中恐怖的服务雪崩问题**，如下图所示：



如上图，这么多服务互相调用，要是不做任何保护的话，某一个服务挂了，就会引起连锁反应，导致别的服务也挂。比如积分服务挂了，会导致订单服务的线程全部卡在请求积分服务这里，没有一个线程可以工作，瞬间导致订单服务也挂了，别人请求订单服务全部会卡住，无法响应。

**但是我们思考一下，就算积分服务挂了，订单服务也可以不用挂啊！为什么？**

我们结合业务来看：支付订单的时候，只要把库存扣减了，然后通知仓库发货就OK了

如果积分服务挂了，大不了等他恢复之后，慢慢人肉手工恢复数据！为啥一定要因为一个积分服务挂了，就直接导致订单服务也挂了呢？不可以接受！

**现在问题分析完了，如何解决？**

这时就轮到Hystrix闪亮登场了。Hystrix是隔离、熔断以及降级的一个框架。啥意思呢？说白了，Hystrix会搞很多个小小的线程池，比如订单服务请求库存服务是一个线程池，请求仓储服务是一个线程池，请求积分服务是一个线程池。每个线程池里的线程就仅仅用于请求那个服务。

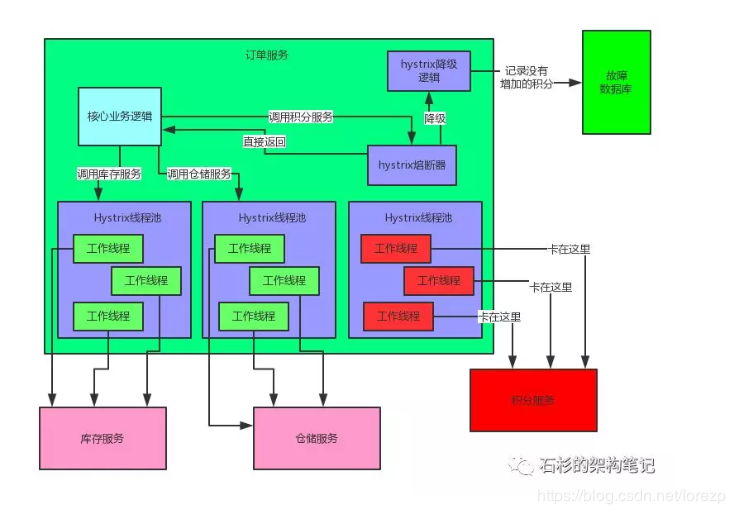
**打个比方：现在很不幸，积分服务挂了，会咋样？**

当然会导致订单服务里的那个用来调用积分服务的线程都卡死不能工作了啊！但是由于订单服务调用库存服务、仓储服务的这两个线程池都是正常工作的，所以这两个服务不会受到任何影响。

这个时候如果别人请求订单服务，订单服务还是可以正常调用库存服务扣减库存，调用仓储服务通知发货。只不过调用积分服务的时候，每次都会报错。**但是如果积分服务都挂了，每次调用都要去卡住几秒钟干啥呢？有意义吗？当然没有！**所以我们直接对积分服务熔断不就得了，比如在5分钟内请求积分服务直接就返回了，不要去走网络请求卡住几秒钟，这个过程，就是所谓的熔断！

**那人家又说，兄弟，积分服务挂了你就熔断，好歹你干点儿什么啊！别啥都不干就直接返回啊？**没问题，咱们就来个降级：每次调用积分服务，你就在数据库里记录一条消息，说给某某用户增加了多少积分，因为积分服务挂了，导致没增加成功！这样等积分服务恢复了，你可以根据这些记录手工加一下积分。这个过程，就是所谓的降级。

为帮助大家更直观的理解，接下来用一张图，梳理一下Hystrix隔离、熔断和降级的全流程：



**Spring Cloud核心组件：Zuul**

说完了Hystrix，接着给大家说说最后一个组件：Zuul，也就是微服务网关。这个组件是负责网络路由的。不懂网络路由？行，那我给你说说，如果没有Zuul的日常工作会怎样？

假设你后台部署了几百个服务，现在有个前端兄弟，人家请求是直接从浏览器那儿发过来的。**打个比方**：人家要请求一下库存服务，你难道还让人家记着这服务的名字叫做inventory-service？部署在5台机器上？就算人家肯记住这一个，你后台可有几百个服务的名称和地址呢？难不成人家请求一个，就得记住一个？你要这样玩儿，那真是友谊的小船，说翻就翻！

上面这种情况，压根儿是不现实的。所以一般微服务架构中都必然会设计一个网关在里面，像android、ios、pc前端、微信小程序、H5等等，不用去关心后端有几百个服务，就知道有一个网关，所有请求都往网关走，网关会根据请求中的一些特征，将请求转发给后端的各个服务。

而且有一个网关之后，还有很多好处，比如可以做统一的降级、限流、认证授权、安全，等等。

### 6. Springboot常见面试问题总结

#### 6.1 什么是springboot

1.用来简化spring应用的初始搭建以及开发过程 使用特定的方式来进行配置（properties或yml文件）  
2.创建独立的spring引用程序 main方法运行  
3.嵌入的Tomcat 无需部署war文件  
4.简化maven配置  
5.自动配置spring添加对应功能starter自动化配置

#### 6.2 springboot常用的starter有哪些

1.spring-boot-starter-web (嵌入tomcat和web开发需要servlet与jsp支持)  
2.spring-boot-starter-data-jpa (数据库支持)  
3.spring-boot-starter-data-redis (redis数据库支持)  
4.spring-boot-starter-data-solr (solr搜索应用框架支持)  
5.mybatis-spring-boot-starter (第三方的mybatis集成starter)

#### 6.3 springboot自动配置的原理

1.@EnableAutoConfiguration这个注释告诉SpringBoot“猜”你将如何想配置Spring,基于你已经添加jar依赖项。如果spring-boot-starter-web已经添加Tomcat和Spring MVC,这个注释自动将假设您正在开发一个web应用程序并添加相应的spring设置。会自动去maven中读取每个starter中的spring.factories文件 该文件里配置了所有需要被创建spring容器中的bean

2.使用@SpringbootApplication注解 可以解决根类或者配置类（我自己的说法，就是main所在类）头上注解过多的问题，一个@SpringbootApplication相当于@Configuration,@EnableAutoConfiguration和 @ComponentScan 并具有他们的默认属性值

#### 6.4 springboot如何添加【修改代码】自动重启功能

添加开发者工具集=====spring-boot-devtools

#### 6.5 Spring Boot 的核⼼心配置⽂文件有哪⼏几 个？它们的区别是什什么？

Spring Boot 的核⼼心配置⽂文件是 application 和 bootstrap 配置⽂文件。 application 配置⽂文件这个容易易理理解，主要⽤用于 Spring Boot 项⽬目的⾃自 动化配置。 bootstrap 配置⽂文件有以下⼏几个应⽤用场景。 使⽤用 Spring Cloud Config 配置中⼼心时，这时需要在 bootstrap

配置⽂文件中添加连接到配置中⼼心的配置属性来加载外部配置中⼼心的 配置信息； ⼀一些固定的不不能被覆盖的属性； ⼀一些加密/解密的场景； 具体请看这篇⽂文章《Spring Boot 核⼼心配置⽂文件详解》。

#### 6.6 Spring Boot 的配置⽂文件有哪⼏几种格 式？它们有什什么区别？

.properties 和 .yml，它们的区别主要是书写格式不不同。 1).properties app.user.name = javastack 2).yml 1. app: 2. user: 3. name: javastack 另外，.yml 格式不不⽀支持 @PropertySource 注解导⼊入配置。

Spring Boot 的核⼼心注解是哪个？它 主要由哪⼏几个注解组成的？

启动类上⾯面的注解是@SpringBootApplication，它也是 Spring Boot 的 核⼼心注解，主要组合包含了了以下 3 个注解： @SpringBootConfiguration：组合了了 @Configuration 注解，实现配置 ⽂文件的功能。 @EnableAutoConfiguration：打开⾃自动配置的功能，也可以关闭某个 ⾃自动配置的选项，如关闭数据源⾃自动配置功能： @SpringBootApplication(exclude = {

DataSourceAutoConfiguration.class })。 @ComponentScan：Spring组件扫描。

#### 6.7 运⾏ Spring Boot 有哪⼏种方式？

1）打包⽤用命令或者放到容器器中运⾏

2)用 Maven/ Gradle 插件运⾏

3)直接执⾏行行 main ⽅方法运⾏

#### 6.8 如何在 Spring Boot 启动的时候运 ⾏行行⼀一些特定的代码？

可以实现接⼝口 ApplicationRunner 或者 CommandLineRunner，这两个 接⼝口实现⽅方式⼀一样，它们都只提供了了⼀一个 run ⽅方法，具体请看这篇⽂文 章《Spring Boot Runner启动器器》。

#### 6.9 Spring Boot 有哪⼏几种读取配置的方式？

Spring Boot 可以通过 @PropertySource,@Value,@Environment, @ConfigurationProperties 来绑定变量量，具体请看这篇⽂文章《Spring Boot读取配置的⼏几种⽅方式》。

#### 6.10 Spring Boot ⽀支持哪些⽇日志框架？ 推荐和默认的⽇日志框架是哪个？

Spring Boot ⽀支持 Java Util Logging, Log4j2, Lockback 作为⽇日志框 架，如果你使⽤用 Starters 启动器器，Spring Boot 将使⽤用 Logback 作为 默认⽇日志框架，具体请看这篇⽂文章《Spring Boot⽇日志集成》。

#### 6.11 SpringBoot 实现热部署有哪⼏几种⽅方 式？

主要有两种⽅方式： Spring Loaded Spring-boot-devtools Spring-boot-devtools 使⽤用⽅方式可以参考这篇⽂文章《Spring Boot实现 热部署》。

#### 6.12 你如何理理解 Spring Boot 配置加载 顺序？

在 Spring Boot ⾥里里⾯面，可以使⽤用以下⼏几种⽅方式来加载配置。

1. properties⽂文件；
2. YAML⽂文件；
3. 系统环境变量量；
4. 命令⾏行行参数； 等等……
5. 具体请看这篇⽂文章《Spring Boot 配置加载顺序详解》。

#### 6.13 Spring Boot 如何定义多套不不同环境 配置？

提供多套配置⽂文件，如：

1. applcation.properties
2. application-dev.properties
3. application-test.properties
4. application-prod.properties 运⾏行行时指定具体的配置⽂文件，

具体请看这篇⽂文章《Spring Boot Profile 不不同环境配置》。

#### 6.14 Spring Boot 可以兼容⽼老老 Spring 项 ⽬目吗，如何做？

可以兼容，使⽤用 @ImportResource 注解导⼊入⽼老老 Spring 项⽬目配置⽂文 件。

#### 6.15 保护 Spring Boot 应⽤用有哪些⽅方 法？

在⽣生产中使⽤用HTTPS 使⽤用Snyk检查你的依赖关系 升级到最新版本 启⽤用CSRF保护 使⽤用内容安全策略略防⽌止XSS攻击 … 更更多请看这篇⽂文章《10 种保护 Spring Boot 应⽤用的绝佳⽅方法》。

#### 6.16 Spring Boot 2.X 有什什么新特性？与 1.X 有什什么区别？

配置变更更 JDK 版本升级 第三⽅方类库升级 响应式 Spring 编程⽀支持 HTTP/2 ⽀支持 配置属性绑定 更更多改进与加强… 具体请看这篇⽂文章《Spring Boot 2.x 新特性总结及迁移指南》。

#### 6.17 Spring Boot ⾃自动配置原理理是什什么？

注解 @EnableAutoConfiguration, @Configuration, @ConditionalOnClass 就是⾃自动配置的核⼼心，⾸首先它得是⼀一个配置⽂文 件，其次根据类路路径下是否有这个类去⾃自动配置。 具体看这篇⽂文章《Spring Boot⾃自动配置原理理、实战》。

### 7. Springcloud常见面试问题总结

#### 7.1 什么是Spring Cloud？

**Spring cloud流应用程序启动器是基于Spring Boot的Spring集成应用程序，提供与外部系统的集成。Spring cloud Task，一个生命周期短暂的微服务框架，用于快速构建执行有限数据处理的应用程序。**

#### 7.2 使用Spring Cloud有什么优势？

**使用Spring Boot开发分布式微服务时，我们面临以下问题**

**1、与分布式系统相关的复杂性-这种开销包括网络问题，延迟开销，带宽问题，安全问题。**

**2、服务发现-服务发现工具管理群集中的流程和服务如何查找和互相交谈。它涉及一个服务目录，在该目录中注册服务，然后能够查找并连接到该目录中的服务。**

**3、冗余-分布式系统中的冗余问题。**

**4、负载平衡 --负载平衡改善跨多个计算资源的工作负荷，诸如计算机，计算机集群，网络链路，中央处理单元，或磁盘驱动器的分布。**

**5、性能-问题 由于各种运营开销导致的性能问题。**

**6、部署复杂性-Devops技能的要求。**

#### 7.3 服务注册和发现是什么意思？Spring Cloud如何实现？

**当我们开始一个项目时，我们通常在属性文件中进行所有的配置。随着越来越多的服务开发和部署，添加和修改这些属性变得更加复杂。有些服务可能会下降，而某些位置可能会发生变化。手动更改属性可能会产生问题。 Eureka服务注册和发现可以在这种情况下提供帮助。由于所有服务都在Eureka服务器上注册并通过调用Eureka服务器完成查找，因此无需处理服务地点的任何更改和处理。**

#### 7.4 负载平衡的意义什么？

**在计算中，负载平衡可以改善跨计算机，计算机集群，网络链接，中央处理单元或磁盘驱动器等多种计算资源的工作负载分布。负载平衡旨在优化资源使用，最大化吞吐量，最小化响应时间并避免任何单一资源的过载。使用多个组件进行负载平衡而不是单个组件可能会通过冗余来提高可靠性和可用性。负载平衡通常涉及专用软件或硬件，例如多层交换机或域名系统服务器进程。**

#### 7.5 什么是Hystrix？它如何实现容错？

**Hystrix是一个延迟和容错库，旨在隔离远程系统，服务和第三方库的访问点，当出现故障是不可避免的故障时，停止级联故障并在复杂的分布式系统中实现弹性。**

**通常对于使用微服务架构开发的系统，涉及到许多微服务。这些微服务彼此协作。**

**思考以下微服务**

**假设如果上图中的微服务9失败了，那么使用传统方法我们将传播一个异常。但这仍然会导致整个系统崩溃。**

**随着微服务数量的增加，这个问题变得更加复杂。微服务的数量可以高达1000.这是hystrix出现的地方 我们将使用Hystrix在这种情况下的Fallback方法功能。我们有两个服务employee-consumer使用由employee-consumer公开的服务。**

**\*\*简化图如下所示 \*\***

**现在假设由于某种原因，employee-producer公开的服务会抛出异常。我们在这种情况下使用Hystrix定义了一个回退方法。这种后备方法应该具有与公开服务相同的返回类型。如果暴露服务中出现异常，则回退方法将返回一些值。**

#### 7.6 什么是Hystrix断路器？我们需要它吗？

**由于某些原因，employee-consumer公开服务会引发异常。在这种情况下使用Hystrix我们定义了一个回退方法。如果在公开服务中发生异常，则回退方法返回一些默认值。**

**如果firstPage method() 中的异常继续发生，则Hystrix电路将中断，并且员工使用者将一起跳过firtsPage方法，并直接调用回退方法。 断路器的目的是给第一页方法或第一页方法可能调用的其他方法留出时间，并导致异常恢复。可能发生的情况是，在负载较小的情况下，导致异常的问题有更好的恢复机会 。**

#### 7.7 什么是Netflix Feign？它的优点是什么？

**Feign是受到Retrofit，JAXRS-2.0和WebSocket启发的java客户端联编程序。Feign的第一个目标是将约束分母的复杂性统一到http apis，而不考虑其稳定性。在employee-consumer的例子中，我们使用了employee-producer使用REST模板公开的REST服务。**

**但是我们必须编写大量代码才能执行以下步骤**

**1、使用功能区进行负载平衡。**

**2、获取服务实例，然后获取基本URL。**

**3、利用REST模板来使用服务。**

**前面的代码如下**

**@Controller**

**public class ConsumerControllerClient {**

**@Autowired**

**private LoadBalancerClient loadBalancer;**

**public void getEmployee() throws RestClientException, IOException {**

**ServiceInstance serviceInstance=loadBalancer.choose("employee-producer");**

**System.out.println(serviceInstance.getUri());**

**String baseUrl=serviceInstance.getUri().toString();**

**baseUrl=baseUrl+"/employee";**

**RestTemplate restTemplate = new RestTemplate();**

**ResponseEntity<String> response=null;**

**try{**

**response=restTemplate.exchange(baseUrl,**

**HttpMethod.GET, getHeaders(),String.class);**

**}catch (Exception ex)**

**{**

**System.out.println(ex);**

**}**

**System.out.println(response.getBody());**

**}**

**之前的代码，有像NullPointer这样的例外的机会，并不是最优的。我们将看到如何使用Netflix Feign使呼叫变得更加轻松和清洁。如果Netflix Ribbon依赖关系也在类路径中，那么Feign默认也会负责负载平衡。**

#### 7.8 什么是Spring Cloud Bus？我们需要它吗？

**考虑以下情况：我们有多个应用程序使用Spring Cloud Config读取属性，而Spring Cloud Config从GIT读取这些属性。**

**下面的例子中多个员工生产者模块从Employee Config Module获取Eureka注册的财产。**

**如果假设GIT中的Eureka注册属性更改为指向另一台Eureka服务器，会发生什么情况。在这种情况下，我们将不得不重新启动服务以获取更新的属性。**

**还有另一种使用执行器端点/刷新的方式。但是我们将不得不为每个模块单独调用这个url。例如，如果Employee Producer1部署在端口8080上，则调用 http：// localhost：8080 / refresh。同样对于Employee Producer2 http：// localhost：8081 / refresh等等。这又很麻烦。这就是Spring Cloud Bus发挥作用的地方。**

**Spring Cloud Bus提供了跨多个实例刷新配置的功能。因此，在上面的示例中，如果我们刷新Employee Producer1，则会自动刷新所有其他必需的模块。如果我们有多个微服务启动并运行，这特别有用。这是通过将所有微服务连接到单个消息代理来实现的。无论何时刷新实例，此事件都会订阅到侦听此代理的所有微服务，并且它们也会刷新。可以通过使用端点/总线/刷新来实现对任何单个实例的刷新。**

# 八.SAAS项目

## 1名词与概念

* SaaS（Software-as-a-Service），即软件即服务。提供给消费者完整的软件解决方案，你可以从软件服务商处以租用或购买等方式获取软件应用，组织用户即可通过 Internet 连接到该应用（通常使用 Web 浏览器）。所有基础结构、中间件、应用软件和应用数据都位于服务提供商的数据中心内。服务提供商负责管理硬件和软件，并根据适当的服务协议确保应用和数据的可用性和安全性。SaaS 让组织能够通过最低前期成本的应用快速建成投产。
* SaaS 软件就适用对象而言，可以划分为针对个人的与针对企业的面向个人的 SaaS 产品：在线文档，账务管理，文件管理，日程计划、照片管理、联系人管理，等等云类型的服务。而面向企业的 SaaS 产品主要包括：CRM（客户关系管理）、ERP（企业资源计划管理）、线上视频或者与群组通话会议、HRM（人力资源管理）、OA(办公系统)、外勤管理、财务管理、审批管理等。
* SaaS与传统软件相比比，a.能够降低企业成本：按需购买，即租即用，无需关注软件的开发维护。b.软件更新迭代快速：和传统软件相比，由于 saas 部署在云端，使得软件的更新迭代速度加快。c.支持远程办公：将数据存储到云后，用户即可通过任何连接到 Internet 的计算机或移动设备访问其信息。
* PaaS（Platform-as-a-Service），即平台即服务。提供给消费者的服务是把客户采用提供的开发语言和工具（例如 Java，python, .Net 等）开发的或收购的应用程序部署到供应商的云计算基础设施上去。客户不需要管理或控制底层的云基础设施，包括网络、服务器、操作系统、存储等，但客户能控制部署的应用程序，也可能控制运行应用程序的托管环境配置。
* IaaS（Infrastructure as a Service），即基础设施即服务。提供给消费者的服务是对所有计算基础设施的利用，包括处理 CPU、内存、存储、网络和其它基本的计算资源，用户能够部署和运行任意软件，包括操作系统和应用程序。消费者不管理或控制任何云计算基础设施，但能控制操作系统的选择、存储空间、部署的应用，也有可能获得有限制的网络组件（例如路由器、防火墙、负载均衡器等）的控制。

## 2系统业务与系统环境

### 2.1业务情况

SaaS物流业务在线上控制线下业务，主要考虑到线下业务周期长，反应慢，沟通效率不高，而当前网络技术发展迅速，网速慢早已不再是限制在线平台发展的主要问题，实时监控线下业务流程也变成了主流的复杂业务解决方案，该项目意在对信息的流转进行全环节控制，服务于货代企业，通过提供接口等工具，帮助中小企业简单便利的对物流全环节沟通、控制等等。项目包括四个方面：

1、 权限管理：对登入系统的员工进行细粒度的权限控制

2、 货物管理：提供货物的全流程管理，包含商品详情，报价

3、 报运管理：包括购销合同，出口报运，装箱，委托，发票

4、 统计管理：以图形化界面的方式对销售，财务数据展示

### 2.2开发环境与技术

项目采用当前流行的前后端分离的编程架构，后端框架采用 Spring +SpringMVC+mybatis +Dubbox，前端采用 AdminLTE 框架。Java采用JDK1.8，数据库使用mysql 5.7，maven 版本为 3.3.9。

#### 2.2.1 SSM创建流程

1.创建 export\_parent 父工程（打包方式选择 pom）

2.创建 export\_web\_manager 子模块（打包方式是 war 包）-负责页面调用等逻辑

3.创建 export\_service\_system 子模块（打包方式是 jar 包）-负责业务逻辑

4.创建 export\_domain 子模块（打包方式是 jar 包）-对应数据库表的pojo

5.创建 export\_dao 子模块（打包方式是 jar 包）-负责数据库连接方面的内容

6.创建 export\_common 子模块（打包方式是 jar 包）-工具包

7.在 export\_parent 的 pom.xml 文件中引入坐标依赖

**2.2.2数据库环境**

* 多租户技术（Multi-TenancyTechnology）又称多重租赁技术：是一种软件架构技术，是实现如何在多用户环境下（此处的多用户一般是面向企业用户）共用相同的系统或程序组件，并且可确保各用户间数据的隔离性。简单讲：在一台服务器上运行单个应用实例，它为多个租户（客户）提供服务。

从定义中我们可以理解：多租户是一种架构，目的是为了让多用户环境下使用同一套程序，且保证用户间数据隔离。那么重点就很浅显易懂了，多租户的重点就是同一套程序下实现多用户数据的隔离。

* 独立数据库：每个租户一个数据库。

优点：为不同的租户提供独立的数据库，有助于简化数据模型的扩展设计，满足不同租户的独特需求；如果出现故障，恢复数据比较简单。

缺点：增多了数据库的安装数量，随之带来维护成本和购置成本的增加这种方案与传统的一个客户、一套数据、一套部署类似，差别只在于软件统一部署在运营商那里。

由此可见此方案用户数据隔离级别最高，安全性最好，但是成本较高

* 共享数据库、独立 Schema：即多个或所有的租户使用同一个数据库服务（如常见的 ORACLE 或 MYSQL 数据库），但是每个租户一个 Schema。

优点： 为安全性要求较高的租户提供了一定程度的逻辑数据隔离，并不是完全隔离；每个数据库可支持更多的租户数量。

缺点： 如果出现故障，数据恢复比较困难，因为恢复数据库将牵涉到其他租户的数据；如果需要跨租户统计数据，存在一定困难。

这种方案是独立数据库的变种。只需要安装一份数据库服务，通过不同的 Schema 对不同租户的数据进行隔离。由于数据库服务是共享的，所以成本相对低廉。

* 共享数据库、共享数据表：即租户共享同一个 Database，同一套数据库表（所有租户的数据都存放在一个数据库的同一套表中）。在表中增加租户 ID 等租户标志字段，表明该记录是属于哪个租户的。

优点：所有租户使用同一套数据库，所以成本低廉。

缺点：隔离级别最低，安全性最低，需要在设计开发时加大对安全的开发量，数据备份和恢复最困难。

这种方案和基于传统应用的数据库设计并没有任何区别，但是由于所有租户使用相同的数据库表，所以需要做好对每个租户数据的隔离安全性处理，这就增加了系统设计和数据管理方面的复杂程度。

* 三范式：

1. 第一范式（1NF ）：确保每一列的原子性（做到每列不可拆分）
2. 第二范式（2NF ）：在第一范式的基础上，非主字段必须依赖于主字段（一个表只做一件事）
3. 第三范式（3NF ）：在第二范式的基础上，消除传递依赖

反三范式：

反三范式是基于第三范式所调整的，没有冗余的数据库未必是最好的数据库，有时为了提高运行效率，就必须降低范式标准，适当保留冗余数据。

### 2.3分页查询业务

#### 2.3.1传统分页

我们之前分页是使用一个 PageBean 对象封装分页数据的。

对象中要定义 currentPage,pageSize,count,totalPage,List<T>等属性，并在 dao 中编写相应的代码查询出分页结果封装到 PageBean 中。

**2.3.2 PageHelper**

PageHelper是国内非常优秀的一款开源的 mybatis 分页插件，它支持基本主流与常用的数据库，例如 mysql、oracle、mariaDB、DB2、SQLite、Hsqldb 等。

* PageHelper.startPage 静态方法调用：在需要进行分页的 MyBatis 查询方法前调用 PageHelper.startPage 静态方法即可，紧跟在这个方法后的第一个 MyBatis 查询方法会被进行分页。

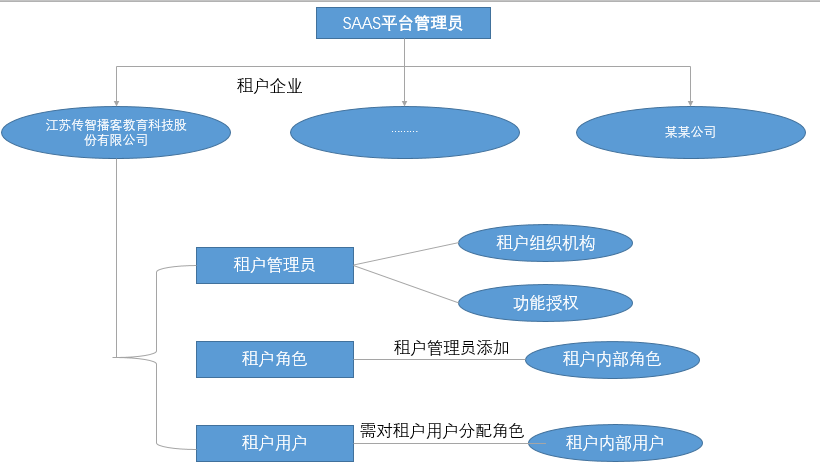
例：//获取第 1 页，10 条内容，默认查询总数 count

PageHelper.startPage(1, 10);

//紧跟着的第一个 select 方法会被分页

List<Country> list = countryMapper.selectIf(1);

**2.4权限**



* SAAS 平台管理员：负责平台的日常维护和管理，包括用户日志的管理、租户账号审核、租户状态管理、租户费用的管理，要注意的是平台管理员不能对租户的具体业务进行管理。
* 企业租户：指访问 SaaS 平台的用户企业，在 SaaS 平台中各租户之间信息是独立的。
* 租户管理员：为租户角色分配权限和相关系统管理、维护。
* 租户用户：需对租户用户进行角色分配，租户用户只能访问授权的模块信息。

**2.5动态构造菜单**



1. 根据登录用户的用户信息查询模块。

2. SaaS 管理员，租户企业管理员直接查询即可。

3. 普通用户，通过用户的角色以及角色和模块之间的关联关系查询。

4. 将模块信息保存到 session 中。

5. 页面动态拼接菜单。

**2.6日志（AOP）**

AOP：面向切面编程。AOP 出现是 OOP 的延伸，是 OOP 的扩展。解决 OOP 中的一些问题。AOP 取代传统的纵向继承体系，采用横向抽取机制。

#### 2.6.1底层原理

Spring实现 AOP 使用两种代理机制：

\* JDK动态代理 :基于接口实现

\* Cglib 动态代理 :基于子类实现

#### 2.6.2注解介绍

@Aspect ：定义切面类

@Before :前置通知

@AfterReturning:后置通知

@Around :环绕通知

@AfterThrowing:异常抛出通知

@After :最终通知

@Pointcut :定义切入点

#### 2.6.3切面类、测试类举例

@Component

@Aspect

public class MyAspect {

@Before("execution(\* cn.itcast.service.CustomerServiceImpl.add\*(..))")

public void before(){

System.out.println("前置增强执行了。。。");

}

}

@Test

public void test1(){

ApplicationContext ac =new ClassPathXmlApplicationContext("applicationContext.xml");

CustomerService cs = (CustomerService) ac.getBean("customerService");

cs.updateCustomer();

}

### 2.7 shiro

#### 2.7.1传统登录方式

步骤如下：

1、Controller 接收请求数据 包含用户名和密码

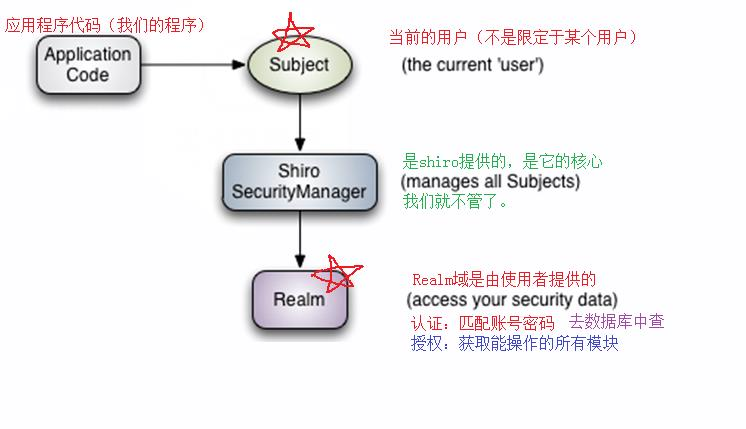
2、调用 Service 去数据库查询 根据 Contoller 提供的用户名和密码

3、判断是否有结果返回

4、如果有返回对象 把返回的用户对象存入 Session 域中

5、如果没有返回对象 直接前往登录页面，并提示登录失败

#### 2.7.2 shiro登录方式



**2.7.3使用步骤**

第一步：在maven父工程中的pom.xml中导入坐标；

第二步：配置spring中提供的shiro器过滤器 （一当十的过滤器）；

第三步：在 spring的配置文件中配置代理方式；

第四步：创建shiro的spring配置文件；

第五步：自定义realm域；

第六步：自定义密码比较器；

第七步：页面使shiro标签，/home/title.jsp主菜单。

**2.7.4 shiro的缓存**

为了提高认证和授权访问的效率，加入shiro自带的Ehcache。shiro每次授权都会通过realm获取权限信息，为了提高访问速度需要添加缓存，第一次从realm中读取权限数据，之后不再读取，这里Shiro和Ehcache整合。引入shiro的core和ehcache Jar包。在shiro的xml配置文件里边引入 缓存的配置。我们在项目中配置的是shiro-ehcache.xml。里边主要配置了一些固定参数，还配置了缓存持久化的目录(序列化地址)。如果配置了缓存，那么在用户修改权限后，要注意清除缓存。

**2.7.5总结**

学习 shiro 框架我们就关心 3 点

1 、Realm 域如何提供

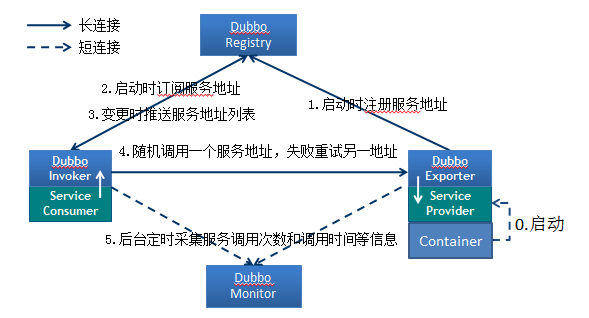
2 、密码比较器如何编写

3 、Action 中的登录方法如何写

### 2.8 Dubbo

我们的项目采用了SOA的架构，利用Duboo+Zookeeper的方式进行不同服务之间的注册以及调用，首先我跟您介绍一下Dubbo，Dubbo致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案。简单的说，dubbo就是个服务框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有dubbo这样的分布式服务框架的需求，并且本质上是个服务调用的东东，说白了就是个远程服务调用的分布式框架。

#### 2.8.1Dubbox的实现原理图如下



#### 2.8.2节点角色说明

Provider: 暴露服务的服务提供方。

Consumer: 调用远程服务的服务消费方。

Registry: 服务注册与发现的注册中心。

Monitor: 统计服务的调用次调和调用时间的监控中心。

Container: 服务运行容器。

#### 2.8.3调用关系说明

0. 服务容器负责启动，加载，运行服务提供者。

1. 服务提供者在启动时，向注册中心注册自己提供的服务。

2. 服务消费者在启动时，向注册中心订阅自己所需的服务。

3. 注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

4. 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

5. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

#### 2.8.4在实际开发的场景中应该如何选择RPC框架

1. **Spring Cloud ：** Spring全家桶，用起来很舒服，只有你想不到，没有它做不到。可惜因为发布的比较晚，国内还没出现比较成功的案例，大部分都是试水，不过毕竟有Spring作背景，还是比较看好。
2. **Dubbox：**相对于Dubbo支持了REST，估计是很多公司选择Dubbox的一个重要原因之一，但如果使用Dubbo的RPC调用方式，服务间仍然会存在API强依赖，各有利弊，懂的取舍吧。
3. **Thrift：** 如果你比较高冷，完全可以基于Thrift自己搞一套抽象的自定义框架吧。
4. **Hessian：**如果是初创公司或系统数量还没有超过5个，推荐选择这个，毕竟在开发速度、运维成本、上手难度等都是比较轻量、简单的，即使在以后迁移至SOA，也是无缝迁移。
5. **rpcx/gRPC：**在服务没有出现严重性能的问题下，或技术栈没有变更的情况下，可能一直不会引入，即使引入也只是小部分模块优化使用。

### 2.9 zookeeper

ZooKeeper是一种为分布式应用所设计的高可用、高性能且一致的开源协调服务，它提供了一项基本服务：**分布式锁服务**。由于ZooKeeper的开源特性，后来我们的开发者在分布式锁的基础上，摸索了出了其他的使用方法：**配置维护、组服务、分布式消息队列、分布式通知/协调**等。

在我们的项目中Zookeeper作为服务注册中心，他的主要任务是负责地址的注册以及查找，相当于一个服务目录，服务的提供者与服务的消费者只在启动时与注册中心进行交互，注册中心也不会转发请求，所以相对而言压力较小。

Zookeeper诞生的背景：

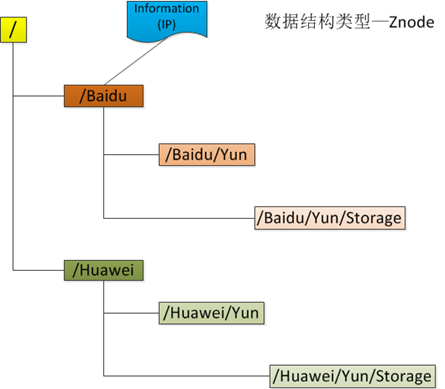
其实除了Zookeeper实现了分布式锁以为还有Google的Chubby也是分布式锁的实现者，Chubby是Googele的闭源程序，所以我们用不了，而Zookeeper是雅虎模仿Chubby开发出来并捐献给Apache的，所以才可以使用到这么优秀的开源程序去构建我们的分布式系统。

#### 2.9.1 Zookeeper是如何实现配置维护、组服务、分布式消息队列等等服务的呢？

首先Zookeeper在实现这些服务之前，他设计了一种新的数据结构-Znode,然后在这个数据结构的基础之上定义了一些对数据结构操作的方法，另外Zookeeper还设计了一种通知机制-Watcher机制。通过新的数据结构-Znode + 操作Znode的方法 + Watcher机制实现了上述我们所看到的服务。

#### 2.9.2简述一下你对Znode的理解。

首先跟你说一下Znode的结构跟我们日常中的Window环境中的文件目录结构很像，下面放张截图

是不是很相似，Zookeeper中的每一个节点（）被称为Znode。

虽然他们很相似，但是他们也有**不同之处。**

**引用方式：**

Znode通过路径进行引用，就像Linux中的文件路径。并且路径都是绝对路径，因此他们都必须用“/”来开头，除此以外他们都必须是唯一的，也就是每一个路径都只有一种表示，并且这些路径不可以改变。在Zookeeper中，这些路径是有Unicode字符串组成，加入一些限制，比如：字符串“/zookeeper”就用以保存管理信息(关键配额信息等)

**Znode结构**

Znode结构兼具了文件和目录两种特点，既可以保存数据、维护数据，又可以向目录一样作为路径表示的一部分。Znode主要是由3个部分组成：

Stat:这个表示状态信息，用于描述这个Znode的版本、权限等信息

Data:这个表示与当前Znode相关联的数据

Children:这个表示当前Znode下的子节点

ZooKeeper虽然可以关联一些数据，但并没有被设计为常规的数据库或者大数据存储，相反的是，它用来管理调度数据，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据的共同特性就是它们都是很小的数据，通常以KB为大小单位。ZooKeeper的服务器和客户端都被设计为严格检查并限制每个Znode的数据大小至多1M，但常规使用中应该远小于此值。

**数据访问**

ZooKeeper中的每个节点存储的数据要被原子性的操作。也就是说读操作将获取与节点相关的所有数据，写操作也将替换掉节点的所有数据。另外，每一个节点都拥有自己的**ACL(访问控制列表)**，这个列表规定了用户的权限，即限定了特定用户对目标节点可以执行的操作。

**节点类型**

ZooKeeper中的节点有两种，分别为临时节点和永久节点。节点的类型在创建时即被确定，并且不能改变。

临时节点：临时节点的生命周期依赖于创建它们的会话。一旦会话(Session)结束，临时节点将被自动删除，当然可以也可以手动删除。虽然每个临时的Znode都会绑定到一个客户端会话，但他们对所有的客户端还是可见的。另外，ZooKeeper的临时节点不允许拥有子节点。

永久节点：永久节点的生命周期不依赖于会话，并且只有在客户端显示执行删除操作的时候，他们才能被删除。

**顺序节点**

当创建Znode的时候，用户可以请求在ZooKeeper的路径结尾添加一个递增的计数。这个计数对于此节点的父节点来说是唯一的，它的格式为"%10d"(10位数字，没有数值的数位用0补充，例如"0000000001")。当计数值大于232-1时，计数器将溢出。

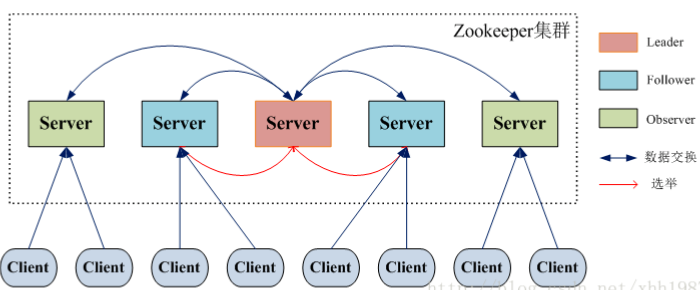
**观察**

客户端可以在节点上设置watch，我们称之为监视器。当节点状态发生改变时(Znode的增、删、改)将会触发watch所对应的操作。当watch被触发时，ZooKeeper将会向客户端发送且仅发送一条通知，因为watch只能被触发一次，这样可以减少网络流量

#### 2.9.3 Zookeeper角色说明

领导者（leader），负责进行投票的发起和决议，更新系统状态  
 学习者（learner），包括跟随者（follower）和观察者（observer），follower用于接受客户端请求并想客户端返回结果，在选主过程中参与投票  
 Observer可以接受客户端连接，将写请求转发给leader，但observer不参加投票过程，只同步leader的状态，observer的目的是为了扩展系统，提高读取速度  
 客户端（client），请求发起方

#### 2.9.4角色调用关系说明：



#### 2.9.5 Zookeeper的核心是什么？

Zookeeper的核心是原子广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。Zab协议有两种模式，它们分别是恢复模式（选主）和广播模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后，恢复模式就结束了。状态同步保证了leader和Server具有相同的系统状态。

#### 2.9.6 Zookeeper中的每个Server有几种状态？

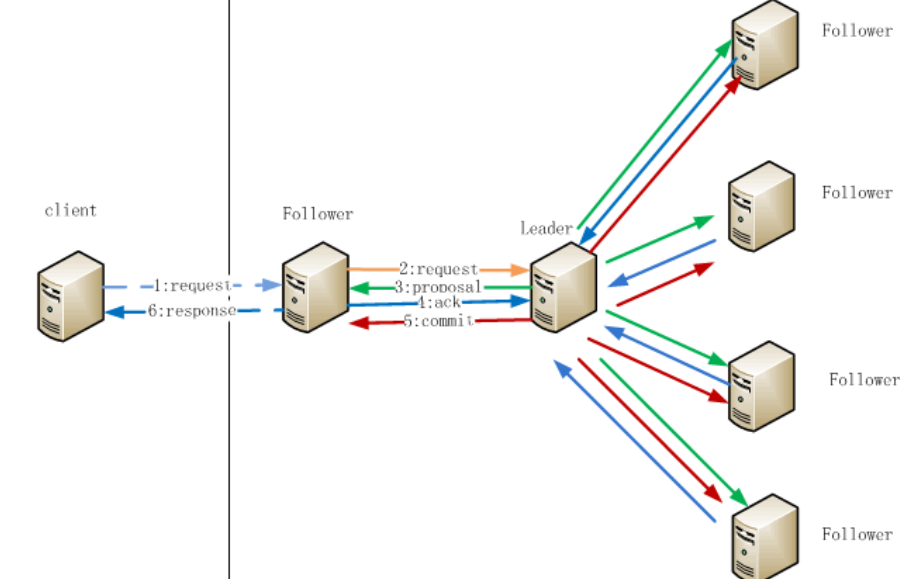
每个Server在工作过程中有三种状态：

LOOKING：当前Server不知道leader是谁，正在搜寻

LEADING：当前Server即为选举出来的leader

FOLLOWING：leader已经选举出来，当前Server与之同步

#### 2.9.7 Zookeeper的节点数据操作流程



1.在Client向Follwer发出一个写的请求

2.Follwer把请求发送给Leader

3.Leader接收到以后开始发起投票并通知Follwer进行投票

4.Follwer把投票结果发送给Leader

5.Leader将结果汇总后如果需要写入，则开始写入同时把写入操作通知给Leader，然后commit;

6.Follwer把请求结果返回给Client

#### 2.9.8为什么zookeeper集群的数目，一般为奇数个？

**1、容错**

由于在增删改操作中需要半数以上服务器通过，来分析以下情况。

**2台服务器**，至少2台正常运行才行（2的半数为1，半数以上最少为2），正常运行1台服务器都不允许挂掉

**3台服务器**，至少2台正常运行才行（3的半数为1.5，半数以上最少为2），正常运行可以允许1台服务器挂掉

**4台服务器**，至少3台正常运行才行（4的半数为2，半数以上最少为3），正常运行可以允许1台服务器挂掉

**5台服务器**，至少3台正常运行才行（5的半数为2.5，半数以上最少为3），正常运行可以允许2台服务器挂掉

**6台服务器**，至少3台正常运行才行（6的半数为3，半数以上最少为4），正常运行可以允许2台服务器挂掉

通过以上可以发现，3台服务器和4台服务器都最多允许1台服务器挂掉，5台服务器和6台服务器都最多允许2台服务器挂掉

但是明显4台服务器成本高于3台服务器成本，6台服务器成本高于5服务器成本。这是由于半数以上投票通过决定的。

**2、防脑裂**

一个zookeeper集群中，可以有多个follower、observer服务器，但是必需只能有一个leader服务器。

如果leader服务器挂掉了，剩下的服务器集群会通过半数以上投票选出一个新的leader服务器。

集群互不通讯情况：

**一个集群3台服务器**，全部运行正常，但是其中1台裂开了，和另外2台无法通讯。3台机器里面2台正常运行过半票可以选出一个leader。

**一个集群4台服务器**，全部运行正常，但是其中2台裂开了，和另外2台无法通讯。4台机器里面2台正常工作没有过半票以上达到3，无法选出leader正常运行。

**一个集群5台服务器**，全部运行正常，但是其中2台裂开了，和另外3台无法通讯。5台机器里面3台正常运行过半票可以选出一个leader。

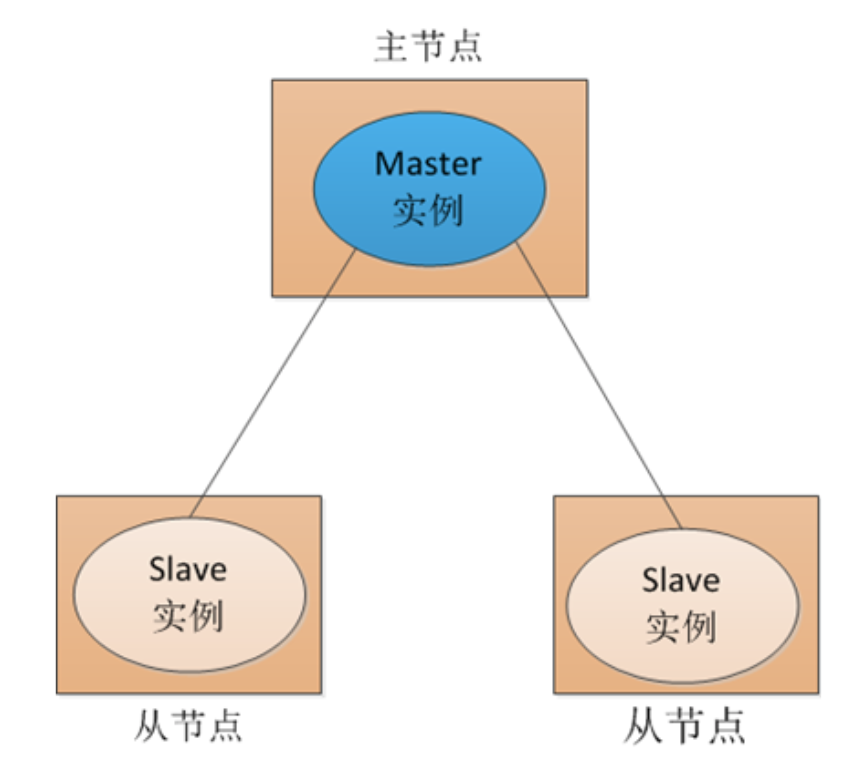
**一个集群6台服务器**，全部运行正常，但是其中3台裂开了，和另外3台无法通讯。6台机器里面3台正常工作没有过半票以上达到4，无法选出leader正常运行。

通可以上分析可以看出，为什么zookeeper集群数量总是单出现，主要原因还是在于第2点，防脑裂，对于第1点，无非是正本控制，但是不影响集群正常运行。但是出现第2种裂的情况，zookeeper集群就无法正常运行了。

#### 2.9.9经典实例：Zookeeper分布式锁的应用场景

**背景：**

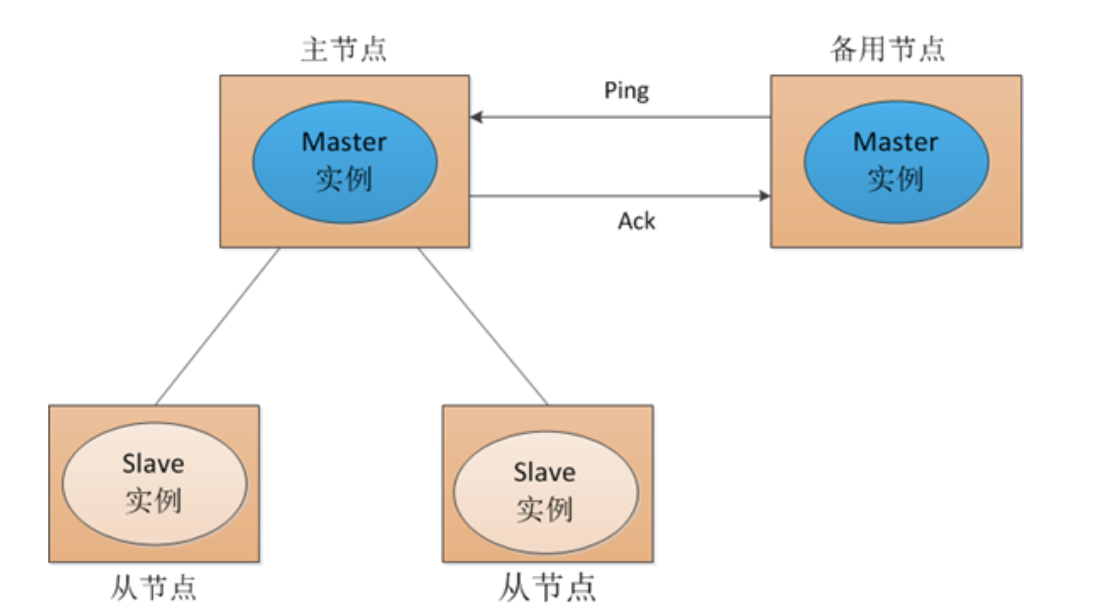
在分布式锁服务中，有一种最典型应用场景，就是**通过对集群进行Master选举**，**来解决**分布式系统中的**单点故障**。什么是分布式系统中的单点故障：通常分布式系统采用主从模式，就是一个主控机连接多个处理节点。主节点负责分发任务，从节点负责处理任务，当我们的主节点发生故障时，那么整个系统就都瘫痪了，那么我们把这种故障叫作单点故障。如下图1.1和1.2所示：

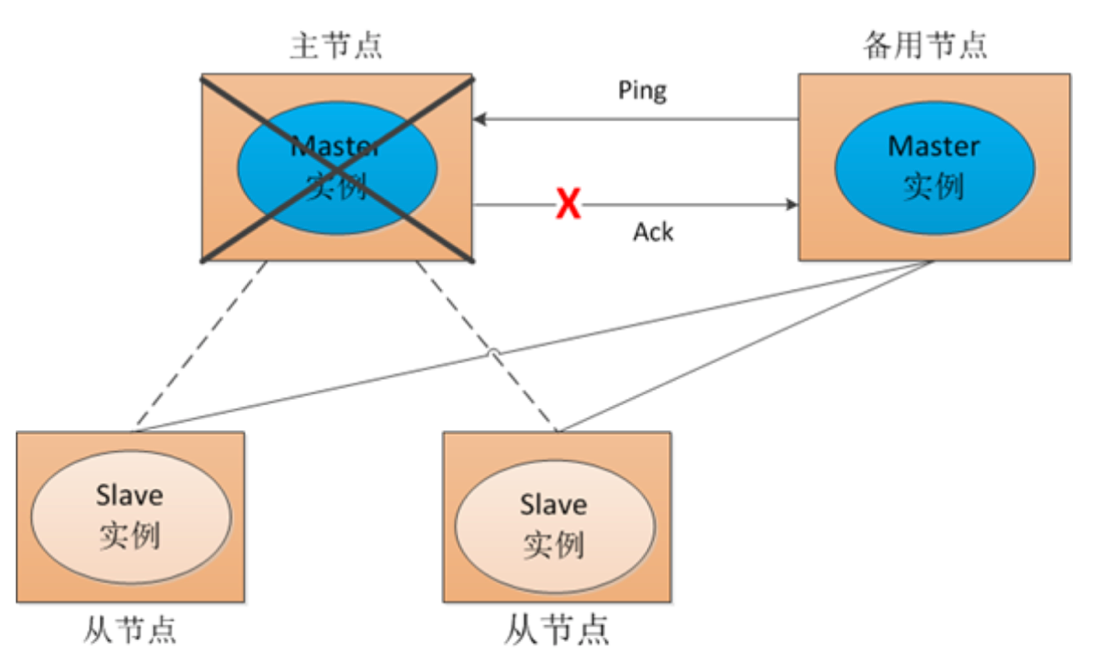
**1.1主从模式分布式系统 1.2单点故障**

**传统解决方案**

传统方式是采用一个**备用节点**，这个备用节点**定期给**当前主节**点发**送**pi**ng包，**主节点收到ping包**以后向备用节**点发**送回复**Ack**，当备用节点收到回复的时候就会认为当前主节点还活着，让他继续提供服务。如图1.3所示

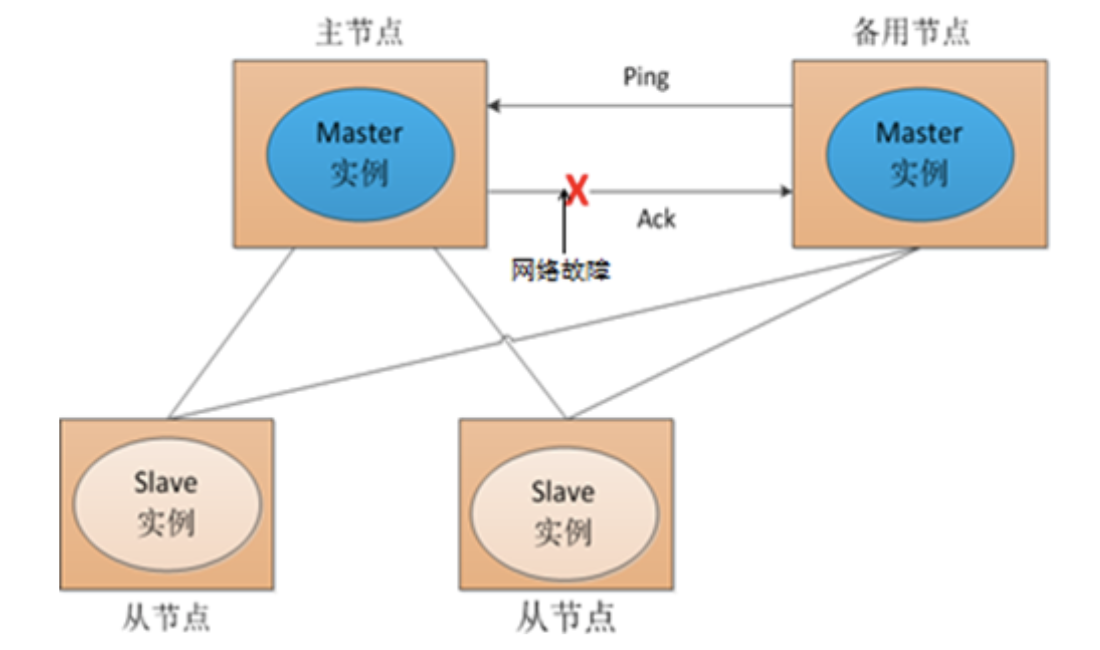


**1.3传统解决方案**

 当主节点挂了，这时候备用节点收不到回复了，然后他就认为主节点挂了接替他成为主节点如下图1.4所示

**1.4传统解决方案**

但是这种方式就是有一个隐患，就是网络问题，来看一网络问题会造成什么后果，如下图1.5所示



**1.5网络故障**

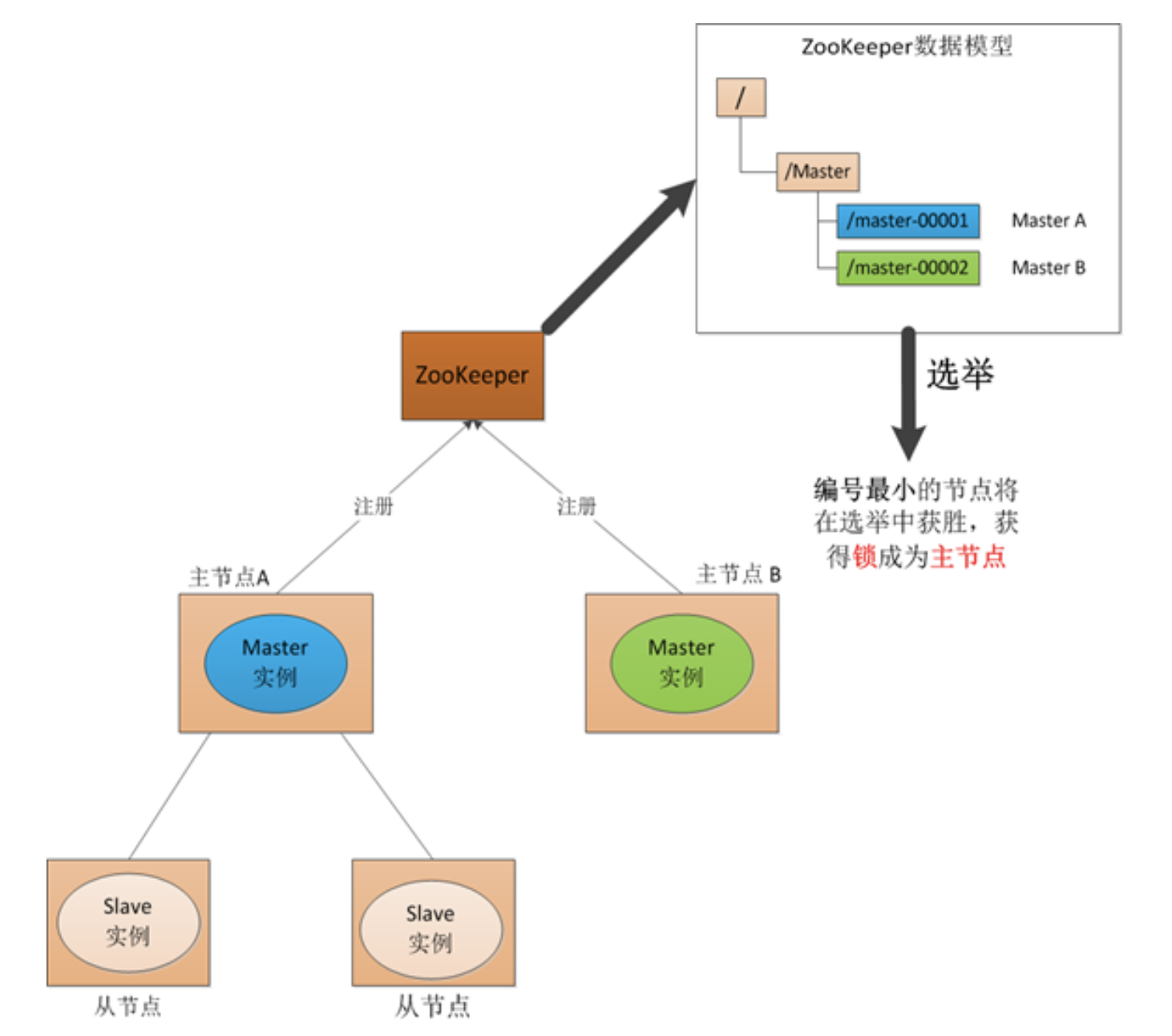
**传统解决方案总结：**

也就是说我们的主节点的并没有挂，只是在回复的时候网络发生故障，这样我们的备用节点同样收不到回复，就会认为主节点挂了，然后备用节点将他的Master实例启动起来，这样我们的分布式系统当中就有了两个主节点也就是---双Master，出现Master以后我们的从节点就会将它所做的事一部分汇报给了主节点，一部分汇报给了从节点，这样服务就全乱了。

**Zookeeper解决方案：**

1. **Master启动：**

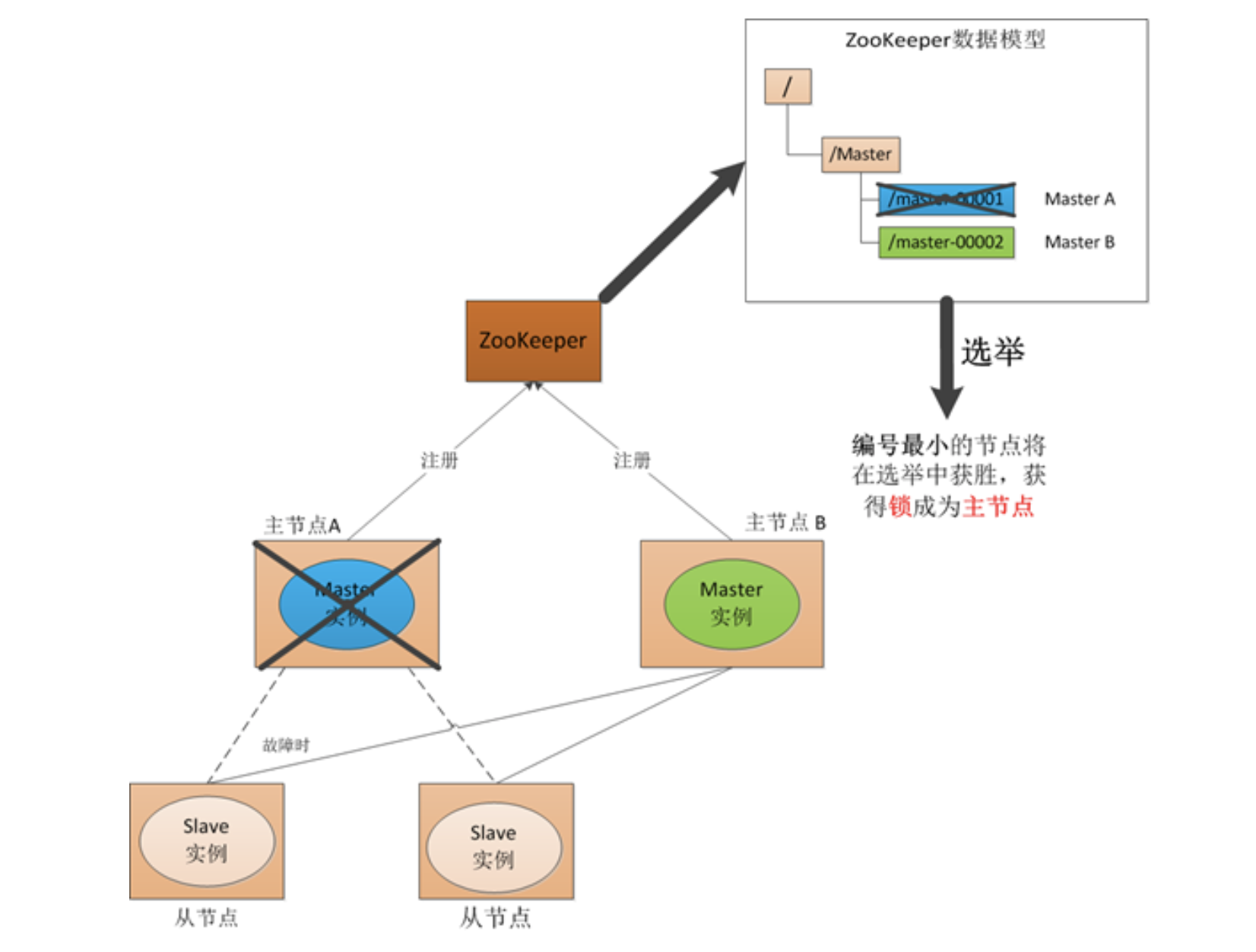
在引入了Zookeeper以后我们启动了两个主节点，"主节点-A"和"主节点-B"他们启动以后，都向ZooKeeper去注册一个节点。我们假设"主节点-A"所注册地节点是"master-00001"，"主节点-B"所注册的节点是"master-00002"，注册完以后进行选举，**编号最小的节点将在选举中获胜**获得**“锁”并且**成为主节点，也就是我们的"主节点-A"将会获得“锁”成为主节点，然后"主节点-B"将被**阻塞**成为一个**备用节点**。那么，通过这种方式就完成了对两个Master进程的调度。如图2.1所示



**2.1 Zookeeper Master选举**

1. **Master故障**

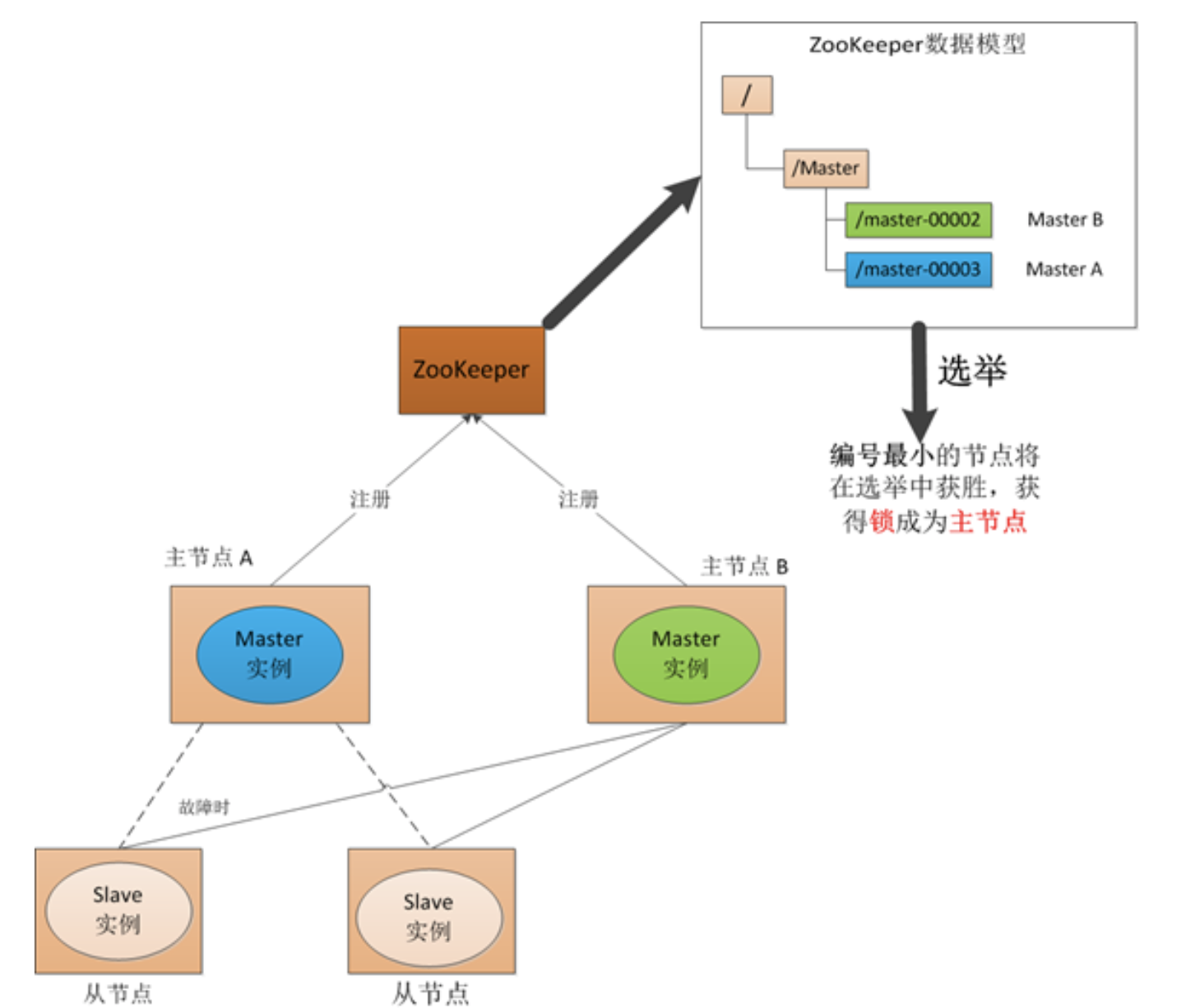
如果**"主节点-A"挂了**，这时候**他**所**注册的节点**将被**自动删除**，**ZooKeeper会**自动感知节点的变化，然后**再次发出选举**，这时候**"主节点-B**"将在选举中**获胜**，替代"主节点-A"**成为主节点。如图2.2所示**

****

**2.2Master故障**

1. **Master恢复**

**如果**“主节点**-A**”**恢复了**，他会**再次**向ZooKeeper**注册**一个**节点**，这时候他注册的节点将会是"**master-00003"**，**ZooKeeper会**感知节点的变化**再次发动选举**，由于**编号最小的节点将在选举中获胜**获得**“锁”并且**成为主节点，这时候"**主节点-B**"在选举中**会再次获胜**继续担任"主节点"，**"主节点-A"会**担任**备用节点**。如图2.3所示：



**2.3Master恢复**

### 2.10 mybatis的逆向工程

#### 2.10.1 mybatis逆向工程概述

mybatis 是目前很流行的持久层框架，很多企业都在采用。但是其复杂繁琐的配置，重复性的实体类创建等等，消耗了程序员大量的精力，同时有些地方如果一个细小的疏忽，可能导致最终功能运行失败。例如：在几十个字段的表中，某一列的列名配置疏忽。基于此，mybatis 推出了一套 jar 包，可以依据我们设计好的数据库表，自动生成 pojo、mapper 以及mapper.xml。有了逆向工程，便大大缩减了我们的开发时间。本章节将介绍借助 idea 的方式实现 mybatis 的逆向工程。

### 2.11 POI技术

POI是用JAVA语言编写的免费的开源的跨平台API，可以对微软的word、Excel、PPT进行操作。

#### 2.11.1 Excel报表导出流程

1. 创建工作簿(HSSFWorkbook)<工作簿这块我们需要注意一个问题，普通处理Excel我们分为Excel03(HSSF)版和07(XSSF)版本以上，对于大数据处理我们使用SXSSF，比如一个Excel有40多列>
2. 创建工作表(createSheet())
3. 设置一些参数(sheet.setColumnWidth(0, 2\*256);)设置列宽，我们一般都会\*256，因为在HSSFWorkbook的底层源码里边对列设置的方法里边，做的处理是二百五十六分之一(1/256)，所以我们要设置列宽的时候，乘以256就是我们想要的值。
4. 根据项目需求我们可能要需要设置标题，设置行高(setHeightInPoints())，设置单元格对象，设置对象里边的内容。还有就是合并单元格，横向合并单元格(sheet.addMergedRegion)等。
5. 把在后台查询的数据填充到对应Excel表格里边。
6. 实现文件下载，导出到制定路径。
7. 根据客户提供的模板和要求做样式的调整，比如：大标题，小标题，字体，居中设置等。

因为不同企业对导出的表格要求不一直，为了便于快速开发，我们在公司一般采取的是模板打印。具体的实现思路是：在后台查询要导入的数据，**制作模版文件（模版文件的路径） 导入（加载）模版文件，从而得到一个工作簿 读取工作表 读取行 读取单元格 读取单元格样式 设置单元格内容 其他单元格就可以使用读到的样式了 。**

针对于Excel的百万级别报表数据导出，我们使用POI提供的SXSSFWork处理大数据量。**但是他不支持模板打印。**

#### 2.11.2百万级数据导出

**针对于大宗贸易我们，涉及的数据量比较大，我们这里采用POI给我们提供的支持07版本的SXSSFWork对象实现导入导出，但是这个也有一个不好的地方就是不能使用模板打印，不能使用太多的样式。需要在后台根据客户需求进行设置。为了比较POI里边的XSSFwork和SXSSFwork我们这里可以借助工具Jvisualvm进行测评在实现大数据量导出的时候，进行系统监控，比如：CPU利用率，gc监控，堆利用率，内存利用率，类监控，线程监控。**

但是在使用SXSSFWork进行导入的时候，因为数据量比较大，经常会造成内存溢出，我们这里使用 POI 基于事件模式解析案例提供的 Excel 文件 。**其实POI里边提供了两种模式，用户模式(系统默认)和事件模式。 这两个的区别是：用户模式一次性加载所有内容到内存，然后着个解析。事件模式是一边加载一边解析。**

**使用这个进行导入时，我们需要定义两个解析器，一个是Excel解析器，这个公司直接封装好的，拿过来直接用。还有一个是sheet的解析器。Sheet解析器是基于SAX解析(**SAX提供了一种从XML文档中读取数据的机制。它逐行扫描文档，一边扫描一边解析。由于应用程序 只是在读取数据时检查数据，因此不需要将数据存储在内存中，这对于大型文档的解析是个巨大优势 **)进行实现的。我们需要在这个解析器里边定义开始行和结束行，创建解析行的方法，在方法里边通过switch {case }语句根据要导入的信息确定值，我们这里用的每一列的表示符(比如：A,B，C等)进行数据处理。**

**原理分析：借助临时存储空间生成 excel** 在实例化 SXSSFWork 这个对象时，可以指定在内存中所产生的 POI 导出相关对象的数量（默认 100），一旦 内存中的对象的个数达到这个指定值时，就将内存中的这些对象的内容写入到磁盘中（XML 的文件格式），就可以 将这些对象从内存中销毁，以后只要达到这个值，就会以类似的处理方式处理，直至 Excel 导出完成。

### 2.12 WebService技术

webService是这项技术指的是在自己网站调用其他网站或者程序。webService指的是一套规范或者服务，主要是为了便于我们跨应用，跨网站进行调用。它约定了不同程序之间的规范，基于xml形式进行数据通信。我们在项目开发中使用的是基于webService的一套框架CXF，Cxf 是基于 SOA 总线结构，依靠 **spring 完成模块的集成**，实现 SOA 方式。 灵活的部署: 可以运行在 Tomcat,Jboss,Jetty(内置),weblogic 上面。

 在工作中，经常会遇到不同公司系统之间的远程服务调用。远程调用技术非常多，如rmi、netty、mina、hessian、dubbo、Motan、springcloud、webservice等等。但是我个人还是比较喜欢webService的，主要原因有4点：

1.webservice技术是建立在http+xml基础之上的，非常的轻量级。

2.webservice技术可通过wsdl来定义调用关系，双方系统可根据wsdl快速的进行开发对接。

3.webservice是一种标准，有各种语言对它的实现，支持异构系统之间的对接。

4.必要情况下，还可以使用httpclient作为客户端进行调用，以降低依赖。

#### 2.12.1 webService的三大规范是什么？

分别指的是JAX-WS，JAX-RS，JAX-WS&SAAJ(废弃)

JAX-RS 是一个 Java 编程语言接口，被设计用来简化使用 REST 架构的应用程序的开发。

借助Jax-RS规范提供一些注解和配置方式：快速的搭建一个符合Restful风格的应用。

借助Jax-RS规范提供的工具类，方便的调用符合Jax-RS的应用。

#### 2.12.2除了webService，还有什么技术可以实现内部程序调用其他网站服务？

目前还有Httpclient、Dubbo<主要应用于内部系统访问>、RMI、Scoket。

**HttpClient**

HttpClient 是 Apache Jakarta Common 下的子项目，用来提供高效的、最新的、功能丰富的支持 HTTP 协议的客户端编程工具包，并且它支持 HTTP 协议最新的版本和建议。主要在调用一些对外的API接口进行使用。

**Web Service**

Web Service提供的服务是基于web容器的，底层使用http协议，类似一个远程的服务提供者，比如天气预报服务，对各地客户端提供天气预报，是一种请求应答的机制，是跨系统跨平台的。就是通过一个servlet，提供服务出去。

首先客户端从服务器的到WebService的WSDL，同时在客户端声称一个代理类(Proxy Class) 这个代理类负责与WebService服务器进行Request 和Response 当一个数据（XML格式的）被封装成SOAP格式的数据流发送到服务器端的时候，就会生成一个进程对象并且把接收到这个Request的SOAP包进行解析，然后对事物进行处理，处理结束以后再对这个计算结果进行SOAP包装，然后把这个包作为一个Response发送给客户端的代理类(Proxy Class)，同样地，这个代理类也对这个SOAP包进行解析处理，继而进行后续操作。这就是WebService的一个运行过程。

Web Service大体上分为5个层次:

 1. Http传输信道

 2. XML的数据格式

 3. SOAP封装格式

 4. WSDL的描述方式 从下往上看这个说明书。

 5. UDDI  UDDI是一种目录服务，企业可以使用它对Webservices进行注册和搜索

#### 2.12.3如何以JAVA的形式启动当前web应用

1.创建工厂

2.制定接口

3.制定交互对象

4.配置日志输出

5.启动服务

#### 2.12.4远程通信的几种选择（RPC，Webservice，RMI，JMS的区别）

RMI （Remote Method Invocation）

RMI 采用stubs 和 skeletons 来进行远程对象(remote object)的通讯。stub 充当远程对象的客户端代理，有着和远程对象相同的远程接口，远程对象的调用实际是通过调用该对象的客户端代理对象stub来完成的，通过该机制RMI就好比它是本地工作，采用tcp/ip协议，客户端直接调用服务端上的一些方法。优点是强类型，编译期可检查错误，缺点是只能基于JAVA语言，客户机与服务器紧耦合。

JMS（Java Messaging Service）

JMS是Java的消息服务，JMS的客户端之间可以通过JMS服务进行异步的消息传输。JMS支持两种消息模型：Point-to-Point（P2P）和Publish/Subscribe（Pub/Sub），即点对点和发布订阅模型。

#### 4521.2.12.5 webService的三要素是什么？

SOAP: 基于 HTTP 协议，采用 XML 格式，用来传递信息的格式。

WSDL: 用来描述如何访问具体的服务。

UDDI: 用户自己可以按 UDDI 标准搭建 UDDI 服务器，用来管理，分发，查询 WebService 。其他用户可以 自己注册发布 WebService 调用。

### 2.13 PDF导出

#### 2.13.1常用的pdf技术有哪些？

1.iText PDF：iText 是著名的开放项目，是用于生成 PDF 文档的一个 java 类库。通过 iText 不仅可以生成 PDF 或 rtf 的文档，而且可以将 XML、Html 文件转化为 PDF 文件。

1. Openoffice：openoffice 是开源软件且能在 windows 和 linux 平台下运行，可以灵活的将 word 或者 Excel 转化为 PDF 文档。
2. **Jasper Report：**是一个强大、灵活的报表生成工具，能够展示丰富的页面内容，并将之转换成 PDF 。

**我们开发中一般选用的是Jasper Report技术，这个技术完全由Java写成，同时还有对应的工具Jaspersoft Studio ，在线编辑很方便。支持多种表格的输出，同时支持多种数据源，通过 JASPER 文件及数据源，JASPER 就能生成最终用户想要的文档格式。**

#### 2.13.2项目中PDF导出的使用

我们在项目中先1.导入坐标，2.引入模板，把编译好的模板引入到当前工程中。3.配置控制器方法(1.加载模板文件，2.构建文件输入流；3.创建 JasperPrint 对象；4.写入 pdf 文档输出 )。导入导出的流程一般都是固定的格式，我自己写了个文档，当我使用的时候，我会拿出来按照上边的步骤进行操作完成。

1.引入 jasper 文件

2.构造数据

a.报运单数据

b.报运货物列表(通过 list 集合创建 javaBean 的数据源对象)

#### 2.13.3简述JasperReport 的生命周期

设计（Design）阶段、执行（Execution）阶段以及输出（Export）阶段。

**设计阶段**就是创建模板，模板创建完成我们保存为JRXML 文件（JR 代表 JasperReports）,其实就是一个 XML 文件。

**执行阶段**编译成可执行的二进制文件(即.Jasper 文件)结合数据进行执行，进行数据填充。

**输出阶段**（Export）：数据填充结束，可以指定输出为多种形式的报表。

#### 2.13.4 JasperReport 的执行流程是什么？

* JRXML:报表填充模板，本质是一个 XML.
* Jasper:由 JRXML 模板编译生成的二进制文件，用于代码填充数据。
* Jrprint:当用数据填充完 Jasper 后生成的文件，用于输出报表。
* Exporter:决定要输出的报表为何种格式，报表输出的管理类。
* Jasperreport 可以输出多种格式的报表文件，常见的有 Html,PDF,xls 等

### 2.14 ActiveMQ

#### 2.14.1消息分发模式有几种？分别有什么特点？

**点对点模式(P2P)和发布-订阅模式(Pub/Sub)。 点对点是**每个消息都被发送到特定的消息队列，接收者从队列中获取消息。队列保留着消息，直到他们被消费或超时。**发布/订阅（Publish-Subscribe）** :包含三个角色：主体（Topic），发布者（Publisher），订阅者（Subscriber）， 多个发布者将消息发送到 topic，系统将这些消息投递到订阅此 topic 的订阅者。

**点对点模型的特点**： 

每个消息只有一个消费者（Consumer）(即一旦被消费，消息就不再在消息队列中)； 

发送者和接收者之间在时间上没有依赖性，也就是说当发送者发送了消息之后，不管接收者有没有正在运行， 它不会影响到消息被发送到队列； 

接收者在成功接收消息之后需向队列应答成功；

**发布/订阅模型的特点：**

每个消息可以有多个消费者；

发布者和订阅者之间有时间上的依赖性。

针对某个主题（Topic）的订阅者，它必须创建一个订阅者之后，才能消费发布者的消息，而且为了消费消息， 订阅者必须保持运行的状态；

#### 2.14.2结合项目聊一聊消息中间件使用场景

我们在项目中主要在新注册用户给用户发邮箱(在用户注册时，因为我们这边会给用户发送邮件，以确定用户是否添加。那么会遇到网络的原因造成用户迟迟收不到邮件，那么我们把发送的事情放到消息队列中。)，到货通知，催单通知。以及电商中的数据同步，比如：数据库和索引库进行同步；页面静态化；

#### 2.14.3 activeMQ收不到消息的原因及解决方式

时间长了就会出现，卡死，新的数据不能从队列接听到。只能重启程序。

解决方案：

1. 不要频繁的建立和关闭连接 JMS 使用长连接方式，一个程序，只要和 JMS 服务器保持一个连接就可以了，不要频繁的 建立和关闭连接。频繁的建立和关闭连接，对程序的性能影响还是很大的。这一点和 jdbc 还是不太一样的。
2. 2）Connection 的 start()和 stop()方法代价很高 JMS 的 Connection 的 start()和 stop()方法代价很高，不能经常调用。我们试用的时候，写了 个 jms 的 connection pool，每次将 connection 取出 pool 时调用 start()方法，归还时调用 stop() 方法，然而后来用 jprofiler 发现，一般的 cpu 时间都耗在了这两个方法上。
3. start()后才能收消息 Connection 的 start()方法调用后，才能收到 jms 消息。如果不调用这个方法，能发出消息， 但是一直收不到消息。不知道其它的 jms 服务器也是这样。 4）显式关闭 Session 如果忘记了最后关闭 Connection 或 Session 对象，都会导致内存泄漏。这个在我测试的时候 也发现了。本来以为关闭了 Connection，由这个 Connection 生成的 Session 也会被自动关闭， 结果并非如此，Session 并没有关闭，导致内存泄漏。所以一定要显式的关闭 Connection 和 Session。
4. 对 Session 做对象池 对 Session 做对象池，而不是 Connection。Session 也是昂贵的对象，每次使用都新建和关闭， 代价也非常高。而且后来我们发现，原来 Connection 是线程安全的，而 Session 不是，所以 后来改成了对 Session 做对象池，而只保留一个 Connection。

#### 2.14.4 activeMQ 存在发生消息太大，造成消息接受不成功

多个线程从 activeMQ 中取消息，随着业务的扩大，该机器占用的网络带宽越来越高。

仔细分析发现，mq 入队时并没有异常高的网络流量，仅仅在出队时会产生很高的网络流量。

最终发现是 spring 的 jmsTemplate 与 activemq 的 prefetch 机制配合导致的问题。

研究源码发现 jmsTemplate 实现机制是：每次调用 receive()时都会创建一个新的 consumer 对象，用完即销毁。

正常情况下仅仅会浪费重复创建 consumer 的资源代价，并不至于产生正常情况十倍百倍的网 络流量。

但是 activeMQ 有一个提高性能的机制 prefetch，此时就会有严重的问题。

prefetch 机制： 每次 consumer 连接至 MQ 时，MQ 预先存放许多 message 到消费者（前提是 MQ 中存在大 量消息），预先存放 message 的数量取决于 prefetchSize（默认为 1000）。此机制的目的很显然，是想让客户端代码用一个 consumer 反复进行 receive 操作，这样能够大量提高出队 性能。

此机制与 jmsTemplate 配合时就会产生严重的问题，每次 jmsTemplate.receive()，都会产 生 1000 个消息的网络流量， 但是因为 jmsTemplae 并不会重用 consumer，导致后面 999 个消息都被废弃。反复 jmsTemplate.receive()时，表面上看 不出任何问题，其实网络带宽会 造成大量的浪费。

解决方案：

1、若坚持使用jmsTemplate，需要设置prefetch值为1，相当于禁用了activeMQ的prefetch 机制，此时感觉最健壮， 就算多线程，反复调用 jmsTemplate.receive()也不会有任何问题。 但是会有资源浪费，因为要反复创建 consumer 并频繁与服务器进 行数据通信，但在性能要求 不高的应用中也不算什么问题。

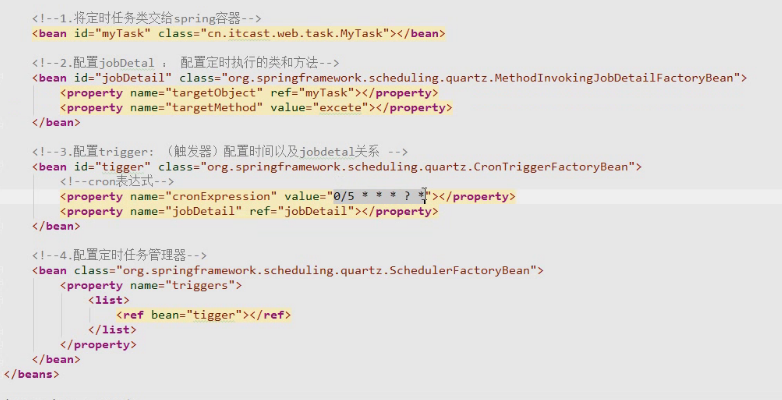
2、不使用 jmsTemplate，手工创建一个 consumer，并单线程反复使用它来 receive()，此 时可以充分利用 prefetch 机制。配合多线程的方式每个线程拥有自己的一个 consumer，此时 能够充分发挥 MQ 在大吞吐量时的速度优势。

切记避免多线程使用一个 consumer 造成的消息混乱。大吞吐量的应用推荐使用方案 2，能够 充分利用 prefetch 机制提高系 MQ 的吞吐性能。

### 2.15 Quartz

1. 项目中的定时任务我们一般使用Quartz或者SpringTask进行实现。我们项目中使用的是Quartz，是一个完全由 Java 编写的开源任务调度的框架，通过触发器设置作业定时运行规则，控制 作业的运行时间。其中 quartz 集群通过故障切换和负载平衡的功能，能给调度器带来高可用性和伸 缩性。主要用来执行定时任务，如：定时发送信息、定时生成报表等等。
2. Quartz在项目中的使用。
   1. 搭建环境(引入Quartz坐标)
   2. 配置一个类，配置定时任务执行的方法
   3. 将此类交给spring进行管理
   4. 配置Quartz和spring整合

我们在开发中经常为如何准确的定位好时间，我们这里可以使用cron.qqe2.com 在线cron表达式生成。Cron他是按照<秒，分钟，小时，日，月，周，年>进行算的。



Quartz和SpringTask的区别是什么？(参考链接：https://blog.csdn.net/swl979623074/article/details/79466016)

实现，Task注解实现方式，比较简单。Quartz需要手动配置Jobs。

任务执行，Task默认单线程串行执行任务，多任务时若某个任务执行时间过长，后续任务会无法及时执行。Quartz采用多线程，无这个问题。

调度，Task采用顺序执行，若当前调度占用时间过长，下一个调度无法及时执行；

Quartz采用异步，下一个调度时间到达时，会另一个线程执行调度，不会发生阻塞问题，但调度过多时可能导致数据处理异常

部署，Quartz可以采用集群方式，分布式部署到多台机器，分配执行定时任务。

# 九.乐优商城

# 十.十次方

## 1．十次方背景

### 11.需求分析：

首先，该项目工程采用的是前后端分离的开发形式。为什么要采用前后端分离的开发形式呢？

《十次方》是程序员的专属社交平台，包括头条、问答、活动、交友、吐槽、招聘六大

频道。

十次方名称的由来:2的10次方为1024，程序员都懂的。

### 1.2.前后端分离的优点和必要性：

前端JS做大部分的数据处理工作，减少对服务器的压力；

后端错误不会直接反映到前台，提高用户体验度；

不同终端（pad/mobile/pc）的兴起，单一浏览器端的访问已经不能满足用户的需求；

JS能够很好的适应前端的效果但是很难做到与服务器的通讯，后端又无法很好的去调整前端页面；

提高开发效率，后端负责业务/数据接口，前端负责展现/交互逻辑，同一份数据接口，可以定制开发多个版本。（一般项目后期会针对性的整理开发接口API方便内外部的调用访问）

### 1.3.什么是前后端分离

经由分析讨论以及资料的收集，我们所一致认同的概念是SPA（single-page application），所有用到的展现数据都是后端通过异步接口（ajax、json）的方式实现的，前端直管展现。

从某种意义上来说，SPA确实做到了前后端分离，但这种方式存在两个问题：

（1） WEB服务中，SPA类占的比例很少。很多场景下还有同步/同步+异步混合的模式，SPA不能作为一种通用的解决方案；

现阶段的SPA开发模式，接口通常是按照展现逻辑来提供的，有时候为了提高效率，后端会帮我们处理一些展现逻辑，这就意味着后端还是涉足了View层的工作，不是真正的前后端分离。

（2）SPA式的前后端分离，是从物理层做区分（认为只要是客户端的就是前端，服务器端的就是后端），这种分法已经无法满足我们前后端分离的需求，我们认为从职责上划分才能满足目前我们的使用场景：

前端：负责View和Controller层；

只负责Model层，业务处理/数据等。

### 1.4.为什么要前后端分离

项目背景：随着互联网的发展，在一定程度上极大的拉近了人与人之间的距离却又在某种程度上增加了人与人之间的隔膜。我们迫切的想要更多人的关注却往往陷入社交平台泛滥的漩涡，基于这种情况我们开发了《十次方》应用于IT方面同学的沟通、交流。让同样的人，说同样的话，得到更多的共鸣与理解。

页面原型的介绍、展示：

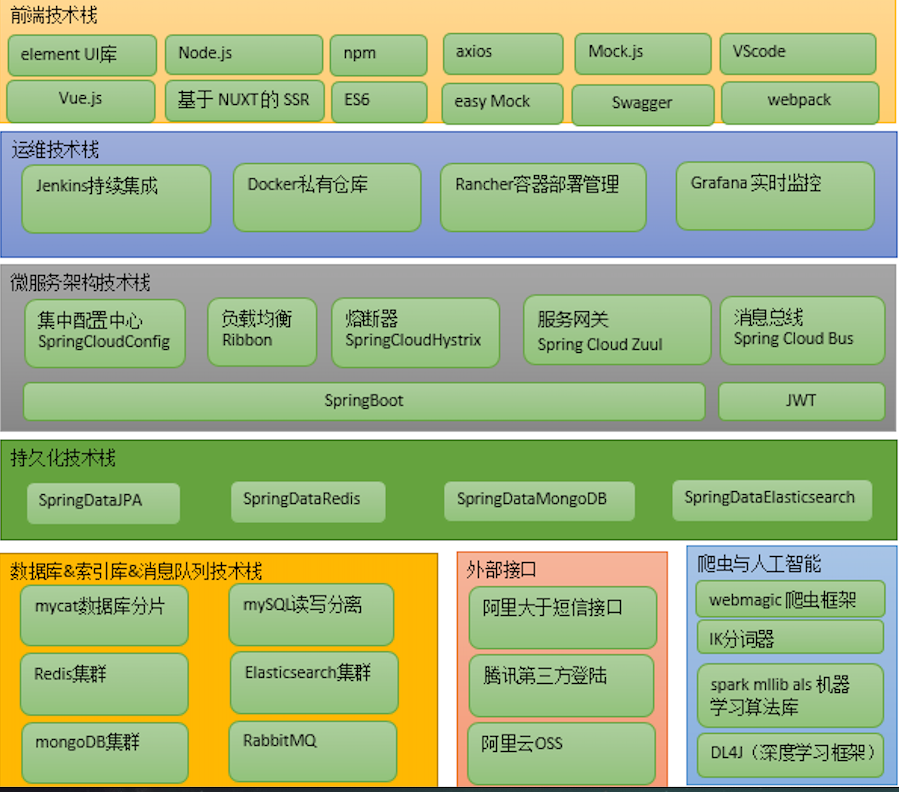


以上，是首页的原型展示。其余界面原型在后面功能实现过程中逐步进行展示。

除此之外，项目开发过程中一般还会有项目需求文档（一般由产品方面来撰写）有的公司可能为了加快开发进度往往先将这部分工作放一放以导致于后期进行项目调整的时候带来诸多屏障。所以，在这一块我还是建议无论项目紧张与否这方面还是要有的。

### 1.5.十次方项目背景：

#### 1.5.1系统架构：



《十次方》采用前后端分离的系统架构，后端架构为:

SpringBoot+SpringCloud+SpringMVC+SpringData

我们把这种架构也称之为全家桶。

#### 1.5.2特色

十次**方采用了当前主流的前后端分离的开发模式，后端使用Spring全家桶框架（即Spring Boot + Spring Cloud + Spring Data + Spring MVC）开发微服务；前端使用以Node.js为核心的Vue全套生态解决方案。项目中涵盖了微服务认证、微服务网关、微服务熔断、微服务集中配置、微服务持续集成、第三方登陆、云存储、爬虫、人工智能、单页面(SPA)、服务端渲染（SSR）等30余种解决方案。**

**采用前后端分离的方式进行系统开发**

采用模块化的课程设计，分为微服务开发，前端系统开发，爬虫与人工智能开发三个模块

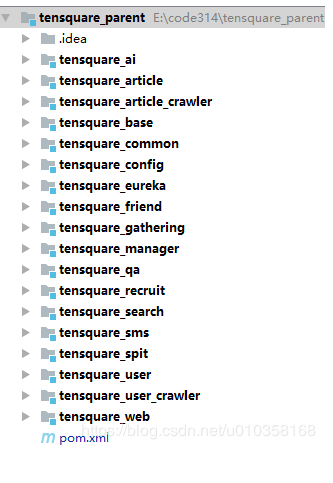
打造Java全栈式工程师,让学员站在Java软件开发的金字塔顶端



#### 1.5.3模块划分

十次方工程共分为18个子模块(其中17个是微服务)。

如下：



模块名称 模块中文名称

tensquare\_common 公共模块

tensquare\_article 文章微服务

tensquare\_ base 基础微服务

tensquare\_friend 交友微服务

tensquare\_gatherinng 活动微服务

tensquare\_qa 问答微服务

tensquare\_recruit 招聘微服务

tensquare\_user 用户微服务

tensquare\_spit 吐槽微服务

tensquare\_search 搜索微服务

tensquare\_web 前台微服务网关

tensquare\_manager 后台微服务网关

tensquare\_eureka 注册中心

tensquare\_config 配置中心

tensquare\_sms 短信微服务

tensquare\_article\_crawler 文章爬虫微服务

tensquare\_user\_crawler 用户爬虫微服务

tensquare\_ai 人工智能微服务

#### 1.5.4表结构分析

文件名 数据库说明

tensquare\_article.sql 文章

tensquare\_ base.sql 基础

tensquare\_friend.sql 交友

tensquare\_gatherinng.sql 活动

tensquare\_qa.sql 问答

tensquare\_recruit.sql 招聘

tensquare\_user.sql 用户



#### 1.5.5API文档

前后端约定返回码列表：

| **状态码** | **说明** |
| --- | --- |
| 20000 | 成功 |
| 20001 | 失败 |
| 20002 | 用户名或密码错误 |
| 20003 | 权限不足 |
| 20004 | 远程调用是白白 |
| 20005 | 重复操作 |

#### 1.5.6开发环境

| **软件** | **版本** |
| --- | --- |
| JDK | 1.8 |
| 数据库 | MySQL 5.7 |
| 开发工具 | IDEA 2017.1 |
| maven | 3.3.9 |
| docker | 1.13.1 |
| 操作系统 | centos 7 |

#### 1.5.7项目主要技术以及功能实现

1.前端环境搭建

章节内容：十次方需求、技术架构，理解前后端分离开发模式, Node.js基本使用方法，理解模块化编程, 包资源管理器NPM的使用, webpack的作用, vs code开发工具的基本使用方法, ES6常用的新特性语法

2.API文档与模拟数据接口

章节内容：RESTful架构, 运用Swagger编写API文档, Mock.js基本语法, easyMock实现模拟接口的编写

3.使用ElementUI开发管理后台

章节内容：elementUI提供的脚手架搭建管理后台的方法, elementUI的table组件的使用和实现列表功能, elementUI的pagination组件的使用和实现分页功能, elementUI的form相关组件的使用和实现条件查询功能, elementUI的dialog组件和$message的使用和实现弹出窗口和消息提示功能, elementUI的select组件的使用和实现下拉列表功能, 新增数据和修改数据的功能, confirm的使用和实现询问与现删除功能

路由与状态管理.

章节内容：路由在单页面工程中的作用, 可搜索下拉框、复合型输入框等ElementUI的使用，完成招聘管理功能, 文章管理功能, Vuex状态管理在工程中的作用

4.网站前台

章节内容：NUXT框架的基本使用方法, 十次方网站前台的搭建, 十次方网站前台活动模块的功能, 十次方网站前台招聘模块的功能, 用户注册功能, js-cookie的使用, 微信扫码登陆的功能, nuxt嵌套路由的使用, 吐槽列表与详细页, 发吐槽与评论功能, 问答频道功能, DataURL和阿里云OSS

5.系统设计与工程搭建

章节内容：十次方的需求分析, 十次方的系统设计, 项目的前期准备工作（配置JDK 与 本地仓库）, 十次方父模块与公共模块的搭建, 基础微服务-标签CRUD的功能，掌握公共异常处理类

6.微服务功能开发

章节内容：基础微服务的开发, 招聘微服务的开发, 问答微服务的开发, 文章微服务的开发, SpringCache与SpringDataRedis的使用

7.文档型数据库MongoDB

章节内容： MongoDb的特点和体系结构, 常用的MongoDB命令,使用运用Java操作MongoDB, 使用SpringDataMongoDB完成吐槽微服务的开发, 使用SpringDataMongoDB完成评论系统的开发

8.消息中间件RabbitMQ与搜索微服务

章节内容：消息队列的应用场景以及RabbitMQ的主要概念, RabbitMQ安装以及RabbitMQ三种模式的入门案例, 用户注册，能够将消息发送给RabbitMQ, 短信微服务，能够接收消息并调用阿里云通信完成短信发送, 搜索微服务

9.密码加密与微服务鉴权JWT

章节内容： BCrypt密码加密算法实现注册与登陆功能, 常见的认证机制, JWT的组成部分以及使用JWT的优点, 使用JJWT 创建和解析token, 使用JJWT完成微服务鉴权

10.SpringCloud初入江湖&SpringCloud一统天下

章节内容： SpringCloud包含的主要框架, SpringCloud包含的主要框架, 使用服务发现组件Eureka, 使用Feign实现服务间的调用, 交友微服务开发, 在项目中使用Hystrix实现微服务熔断, 在项目中使用Zuul实现微服务网关, 在项目中使用SpringCloudConfig实现配置集中管理, 在项目中使用SpringCloudBus实现配置的在线更新

11.爬虫

章节内容：什么是网络爬虫, 网络爬虫可以做什么, 网络爬虫常用的技术, 爬虫框架Webmagic, 十次方文章数据爬取, 十次方用户数据爬取,

12.人工智能

章节内容：学习目标, 人工智能与机器学习, 智能分类, IK分词器, Deeplearning4j之Word2VEC, 构建卷积神经网络模型, 实现智能分类

#### 1.5.8restful

RESTful架构，就是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易 于理解、扩展方便，所以正得到越来越多网站的采用。REST这个词，是Roy Thomas Fielding在他2000年的博士论文中提出的。

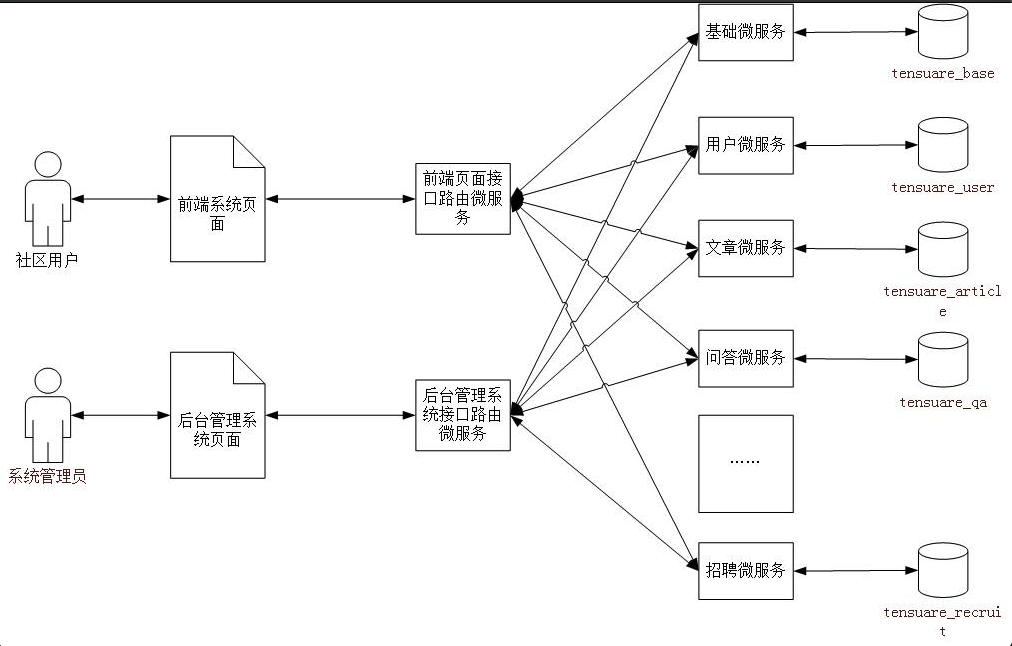
REST是Representational State Transfer的缩写，翻译是”表现层状态转化”。

可以 总结为一句话:REST是所有Web应用都应该遵守的架构设计指导原则。

面向资源是REST最明显的特征，对于同一个资源的一组不同的操作。资源是服务器 上一个可命名的抽象概念，资源是以名词为核心来组织的，首先关注的是名词。REST要求，必须通过统一的接口来对资源执行各种操作。对于每个资源只能执行一组有限的操 作。

7个HTTP方法：GET/POST/PUT/DELETE/PATCH/HEAD/OPTIONS。

## 十次方技术架构



## 3．十次方面试中常遇到的问题：

### 3.1对微服务有何了解？

微服务，又称微服务架构，是一种架构风格，它将应用程序构建为以业务领域为模型的小型自治服务集合 。  
通俗的来说就是将业务模块区分，不受编程语言限制，提供更好的可扩展性，独立开发部署。一个业务模块的改动不会影响到其他业务。可以单独处理每个服务组件的问题，而对整个应用程序没有影响或影响最小。

### 3.2 微服务架构有哪些优势？

* **独立开发** – 所有微服务都可以根据各自的功能轻松开发
* **独立部署** – 基于其服务，可以在任何应用程序中单独部署它们
* **故障隔离** – 即使应用程序的一项服务不起作用，系统仍可继续运行
* **混合技术堆栈** – 可以使用不同的语言和技术来构建同一应用程序的不同服务
* **粒度缩放** – 单个组件可根据需要进行缩放，无需将所有组件缩放在一起

### 3.3微服务有哪些特点？

* **解耦** – 系统内的服务很大程度上是分离的。因此，整个应用程序可以轻松构建，更改和扩展
* **组件化** – 微服务被视为可以轻松更换和升级的独立组件
* **业务能力** – 微服务非常简单，专注于单一功能
* **自治** – 开发人员和团队可以彼此独立工作，从而提高速度
* **持续交付** – 通过软件创建，测试和批准的系统自动化，允许频繁发布软件
* **责任** – 微服务不关注应用程序作为项目。相反，他们将应用程序视为他们负责的产品
* **分散治理** – 重点是使用正确的工具来做正确的工作。这意味着没有标准化模式或任何技术模式。开发人员可 以自由选择最有用的工具来解决他们的问题  
  **敏捷** – 微服务支持敏捷开发。任何新功能都可以快速开发并再次丢弃

### 3.4 微服务架构如何运作？

* **客户端** – 来自不同设备的不同用户发送请求。
* **身份提供商** – 验证用户或客户身份并颁发安全令牌。
* **API网关** – 处理客户端请求。
* **静态内容** – 容纳系统的所有内容。
* **管理** – 在节点上平衡服务并识别故障。
* **服务发现** – 查找微服务之间通信路径的指南。
* **内容交付网络** – 代理服务器及其数据中心的分布式网络。
* **远程服务** – 启用驻留在IT设备网络上的远程访问信息。

### 3.5 微服务架构的优缺点是什么？

| **微服务架构的优点** | **微服务架构的缺点** |
| --- | --- |
| 自由使用不同的技术 | 增加故障排除挑战 |
| 每个微服务都侧重于单一功能 | 由于远程呼叫而增加延迟 |
| 支持单个可部署单元 | 增加了配置和其他操作的工作量 |
| 允许经常发布软件 | 难以保持交易安全 |
| 确保每项服务的安全性 | 艰难地跨越各种边界跟踪数据 |
| 多个服务是并行开发和部署的 | 难以在服务之间进行编码 |

### 3.6 单片，SOA和微服务架构有什么区别？

* **单片架构**类似于大容器，其中应用程序的所有软件组件组装在一起并紧密封装。
* 一个**面向服务**的架构是一种相互通信服务的集合。通信可以涉及简单的数据传递，也可以涉及两个或多个协调某些活动的服务。
* **微服务**架构是一种架构风格，它将应用程序构建为以业务域为模型的小型自治服务集合。

### 3.7 在使用微服务架构时，您面临哪些挑战？

开发一些较小的微服务听起来很容易，但开发它们时经常遇到的挑战如下。

* **自动化组件**：难以自动化，因为有许多较小的组件。因此，对于每个组件，我们必须遵循Build，Deploy和Monitor的各个阶段。
* **易感性**：将大量组件维护在一起变得难以部署，维护，监控和识别问题。它需要在所有组件周围具有很好的感知能力。
* **配置管理**：有时在各种环境中维护组件的配置变得困难。
* **调试**：很难找到错误的每一项服务。维护集中式日志记录和仪表板以调试问题至关重要。

### 3.8 SOA和微服务架构之间的主要区别是什么？

SOA和微服务之间的主要区别如下：

| **SOA** | **微服务** |
| --- | --- |
| 遵循“ 尽可能多的共享 ”架构方法 | 遵循“ 尽可能少分享 ”的架构方法 |
| 重要性在于 业务功能 重用 | 重要性在于“ 有界背景 ” 的概念 |
| 他们有 共同的 治理 和标准 | 他们专注于 人们的 合作 和其他选择的自由 |
| 使用 企业服务总线（ESB） 进行通信 | 简单的消息系统 |
| 它们支持 多种消息协议 | 他们使用 轻量级协议 ，如 HTTP / REST 等。 |
| 多线程， 有更多的开销来处理I / O. | 单线程 通常使用Event Loop功能进行非锁定I / O处理 |
| 最大化应用程序服务可重用性 | 专注于 解耦 |
| 传统的关系数据库 更常用 | 现代 关系数据库 更常用 |
| 系统的变化需要修改整体 | 系统的变化是创造一种新的服务 |
| DevOps / Continuous Delivery正在变得流行，但还不是主流 | 专注于DevOps /持续交付 |

### 3.9 什么是耦合？

组件之间依赖关系强度的度量被认为是**耦合**。一个好的设计总是被认为具有**高内聚力**和**低耦合性**。

### 3.10什么是REST / RESTful以及它的用途是什么？

（REST）/ RESTful Web服务是一种帮助计算机系统通过Internet进行通信的架构风格。这使得微服务更容易理解和实现。  
微服务可以使用或不使用RESTful API实现，但使用RESTful API构建松散耦合的微服务总是更容易。

### 3.11 你对Spring Boot有什么了解？

事实上，随着新功能的增加，弹簧变得越来越复杂。如果必须启动新的spring项目，则必须添加构建路径或添加maven依赖项，配置应用程序服务器，添加spring配置。所以一切都必须从头开始。  
Spring Boot是解决这个问题的方法。使用spring boot可以避免所有样板代码和XML配置。快速构建项目。感觉是springMVC精简版。

### 3.12 什么是Spring引导的执行器？

Spring Boot执行程序提供了restful Web服务，以访问生产环境中运行应用程序的当前状态。在执行器的帮助下，您可以检查各种指标并监控您的应用程序。

### 3.13 什么是Spring Cloud？

根据Spring Cloud的官方网站，Spring Cloud为开发人员提供了快速构建分布式系统中一些常见模式的工具（例如配置管理，服务发现，断路器，智能路由，领导选举，分布式会话，集群状态）。

### 3.14 Spring Cloud解决了哪些问题？

在使用Spring Boot开发分布式微服务时，我们面临的问题很少由Spring Cloud解决。

* **与分布式系统相关的复杂性** – 包括网络问题，延迟开销，带宽问题，安全问题。
* **处理服务发现的能力** – 服务发现允许集群中的进程和服务找到彼此并进行通信。
* **解决冗余问题** – 冗余问题经常发生在分布式系统中。
* **负载平衡** – 改进跨多个计算资源（例如计算机集群，网络链接，中央处理单元）的工作负载分布。
* **减少性能问题** – 减少因各种操作开销导致的性能问题。

### 3.15 在Spring MVC应用程序中使用WebMvcTest注释有什么用处？

  
在测试目标只关注Spring MVC组件的情况下，**WebMvcTest**注释用于单元测试Spring MVC应用程序。在上面显示的快照中，我们只想启动ToTestController。执行此单元测试时，不会启动所有其他控制器和映射。

### 3.16 你能否给出关于休息和微服务的要点？

休息

虽然您可以通过多种方式实现微服务，但REST over HTTP是实现微服务的一种方式。REST还可用于其他应用程序，如Web应用程序，API设计和MVC应用程序，以提供业务数据。

微服务

微服务是一种体系结构，其中系统的所有组件都被放入单独的组件中，这些组件可以单独构建，部署和扩展。微服务的某些原则和最佳实践有助于构建弹性应用程序。

简而言之，您可以说REST是构建微服务的媒介。

### 3.17 什么是不同类型的微服务测试？

在使用微服务时，由于有多个微服务协同工作，测试变得非常复杂。因此，测试分为不同的级别。

* 在**底层**，我们有面向技术的测试，如单元测试和性能测试。这些是完全自动化的。
* 在**中间层面**，我们进行了诸如压力测试和可用性测试之类的探索性测试。
* 在**顶层**， 我们的 验收测试数量很少。这些验收测试有助于利益相关者理解和验证软件功能。

### 3.18 您对Distributed Transaction有何了解？

**分布式事务**是指单个事件导致两个或多个不能以原子方式提交的单独数据源的突变的任何情况。在微服务的世界中，它变得更加复杂，因为每个服务都是一个工作单元，并且大多数时候多个服务必须协同工作才能使业务成功。

### 3.19 什么是Idempotence以及它在哪里使用？

**幂等**性是能够以这样的方式做两次事情的特性，即最终结果将保持不变，即好像它只做了一次。

**用法**：在远程服务或数据源中使用 Idempotence，这样当它多次接收指令时，它只处理指令一次。

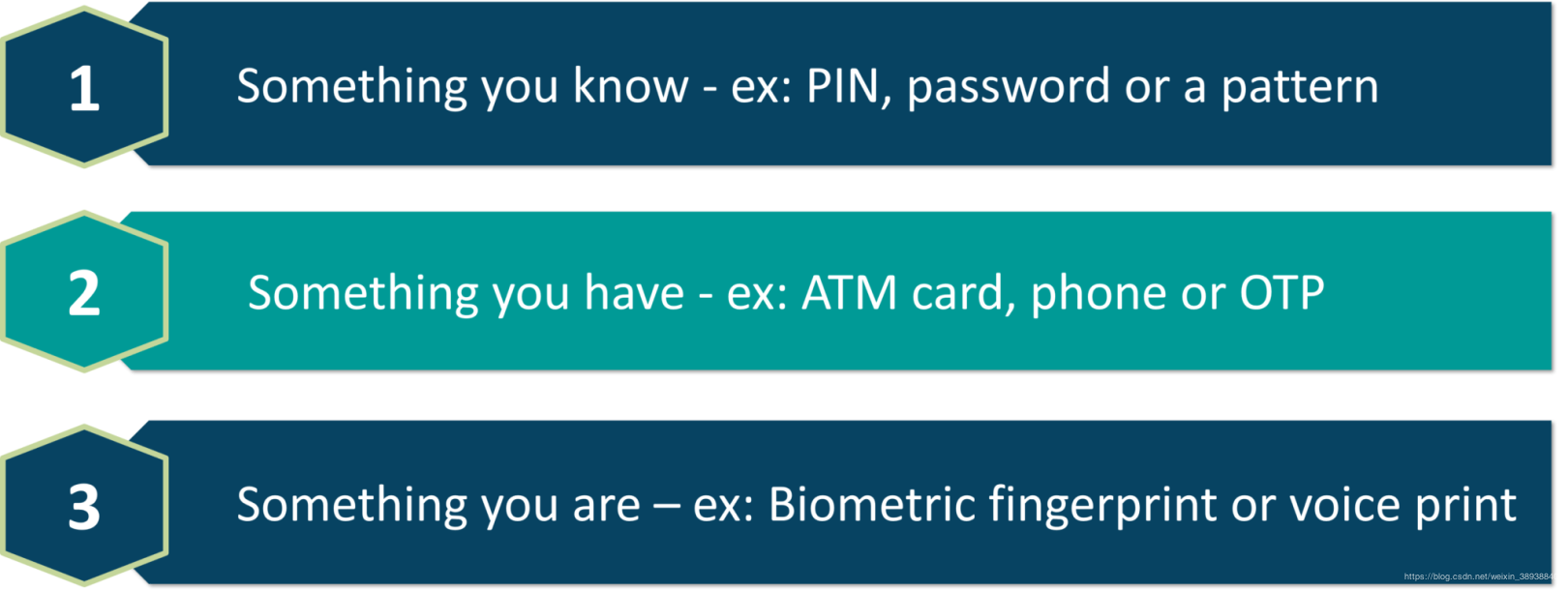
### 3.20 什么是有界上下文？

有界上下文是域驱动设计的核心模式。DDD战略设计部门的重点是处理大型模型和团队。DDD通过将大型模型划分为不同的有界上下文并明确其相互关系来处理大型模型。

### 3.21什么是双因素身份验证？

双因素身份验证为帐户登录过程启用第二级身份验证。  
因此，假设用户必须只输入用户名和密码，那么这被认为是单因素身份验证。

### 3.22 双因素身份验证的凭据类型有哪些？



### 3.23 什么是客户证书？

客户端系统用于向远程服务器发出经过身份验证的请求的一种数字证书称为**客户端证书**。客户端证书在许多相互认证设计中起着非常重要的作用，为请求者的身份提供了强有力的保证。比如JWT鉴权等。。